MASTER EN SCIENCES INFORMATIQUES

Animation graphique de tâches de bureau

Implémentation en Smalltalk-80 d'un outils d'aide à l'automatisation du travail de bureau

Provot, Isabelle; Gendarme, Philippe

Award date:
1988

Awarding institution:
Universite de Namur

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX - NAMUR

INSTITUT D'INFORMATIQUE

ANIMATION GRAPHIQUE DE TACHES DE BUREAU

Implémentation en Smalltalk-80
d'un outil d'aide à
l'automatisation du travail de bureau

Isabelle Provot Philippe Gendarme

Mémoire présenté en vue de l'obtention

du grade de Licencié et Maître en Informatique

par

Isabelle Provot

Philippe Gendarme

PROMOTEUR:
Monsieur Roland Lesuisse

Année Académique 1987 - 1988

Remerciements

A Monsieur Roland Lesuisse, promoteur de ce mémoire, nous adressons nos remerciements les plus chaleureux pour le stage qu'il nous a procuré, l'aide qu'il nous a apportée et pour les encouragements sans cesse renouvelés qu'il nous a prodigués tout au long de l'élaboration de notre mémoire.

Nous exprimons également notre plus vive gratitude à Monsieur Dominique Genin, Manager du Tektronix A.I. Research Center de Leuven, et à ses collaborateurs Mademoiselle Anne Dardenne et Monsieur Jan De Moortel pour leurs judicieux et précieux conseils lors de notre apprentissage du système Smalltalk-80.

Nous remercions Madame Nadine Dachouffe-Jamotte pour ses remarques pertinentes lors de la lecture d'une version préliminaire de notre mémoire et pour avoir mis à notre disposition sa documentation et ses travaux personnels qui sont à la base de notre travail.

Enfin, nous ne voulons pas oublier toutes les personnes qui nous ont aidés de près ou de loin lors de nos travaux de recherche et de rédaction.

Facultés Universitaires Notre-Dame de la Paix Namur
Institut d'informatique

Animation graphique de tâches de bureau

Implémentation en Smalltalk-80 d'un outil d'aide à l'automatisation du travail de bureau

Isabelle Provot Philippe Gendarme

Résumé:

L'automatisation du travail de bureau rencontre des problèmes dûs à la difficulté qu'éprouvent l'analyste et l'utilisateur à communiquer. Ce mémoire présente un outil dont le principe de base est la simulation graphique des tâches de bureau à automatiser et dont l'utilisation permet d'améliorer le dialogue entre les personnes impliquées. Nous y décrivons le cadre général de l'automatisation et les méthodes principalement utilisées, les exigences auxquelles doit satisfaire l'outil proposé parmi lesquelles son extensibilité et sa flexibilité. Une partie importante de ce mémoire est consacrée à la présentation des moyens de réalisation à savoir la programmation orientée objet et l'environnement Smalltalk-80, et à la description de l'implémentation de l'outil.

Abstract:

Currently, office automation encounters problems due to difficulties in the dialog between analysts and users. This thesis presents a tool which is based on the graphical simulation of office tasks to automate, and whose use allows improved talk between involved persons. We describe the general framework of automation and the principally used methods, requirements to which must satisfy the proposed tool among which its entensibility and flexibility. An important part of this thesis is dedicated to the implementation means - the oriented object programming and the Smalltalk-80 environment - and to the description of the implementation of the tool.

Mémoire de licence et maîtrise en informatique

Septembre 1988

Promoteur: Monsieur Roland Lesuisse

TABLE DES MATIERES

CHAPITRE 1: OBJECTIFS ET CADRE DU TRAVAIL

CHAPITRE 2: ETUDES ANTERIEURES ET CRITIQUES

CHAPITRE 1

OBJECTIFS ET CADRE DU TRAVAIL

CHAPITRE 1: OBJECTIFS ET CADRE DU TRAVAIL

Depuis de nombreuses années, l'informatique et les technologies qui en dérivent, s'introduisent à grands pas dans le monde des bureaux. Au début des années 80, l'informatisation croissante des activités de bureau fut baptisée *bureautique* . Cependant, la signification exacte de ce terme couramment utilisé, est encore difficile à cerner. Nous nous en tiendrons, pour notre part, à la définition de de Blasis [BLAS-82] qui met en évidence les diverses composantes dont la composante humaine.

> "La bureautique s'inscrit dans le cadre de l'informatisation progressive de notre société en s'appliquant spécifiquement aux activités de bureau. Il ne s'agit pas d'une révolution fracassante de nos méthodes de travail dans les bureaux, mais d'une évolution adaptant les moyens technologiques modernes aux multiples tâches que nous y accomplissons. L'objectif est d'améliorer la qualité et la rapidité du travail de bureau, son efficience voire sa productivité en tirant parti des possiblités de l'électronique, de l'informatique, des télécommunications, etc." mais "Avant toute chose, *l'optique de la bureautique est de privilégier l'homme sur les machines* ".

C'est dans ce contexte général de la bureautique et de son intérêt pour les facteurs humains et sociaux que prend place notre mémoire. Il s'insère dans un plus vaste projet visant à la réalisation d'un logiciel d'aide à la conception de systèmes d'information de bureau.

1.1 L'environnement de développement

Afin de se faire une idée précise des objectifs du projet, il est primordial de situer le cadre général dans lequel se déroule tout processus d'automatisation du travail de bureau, et d'identifier les facteurs qui peuvent influencer le développement d'un pareil système d'information.

Etant donné la complexité des organisations et, plus particulièrement des bureaux, toute conception d'un système d'information se base sur un modèle qui sert généralement de support à l'étude du développement, de l'utilisation ou de l'impact d'un système d'information dans une organisation. Parmi ces divers modèles, figure celui de Ives [IVES-80].

Lorsque l'on examine ce modèle, on s'aperçoit que la majeure partie de ses composants provient d'un certain nombre d'environnements dont les caractéristiques influencent le système d'information.

Ce modèle met en évidence trois types d'environnements, trois types de processus liés au système d'information et le sous-système d'information lui-même. Tous ces éléments sont inclus dans un environnement organisationnel et un environnement externe. Ce modèle peut être schématisé ainsi (figure 1.1):

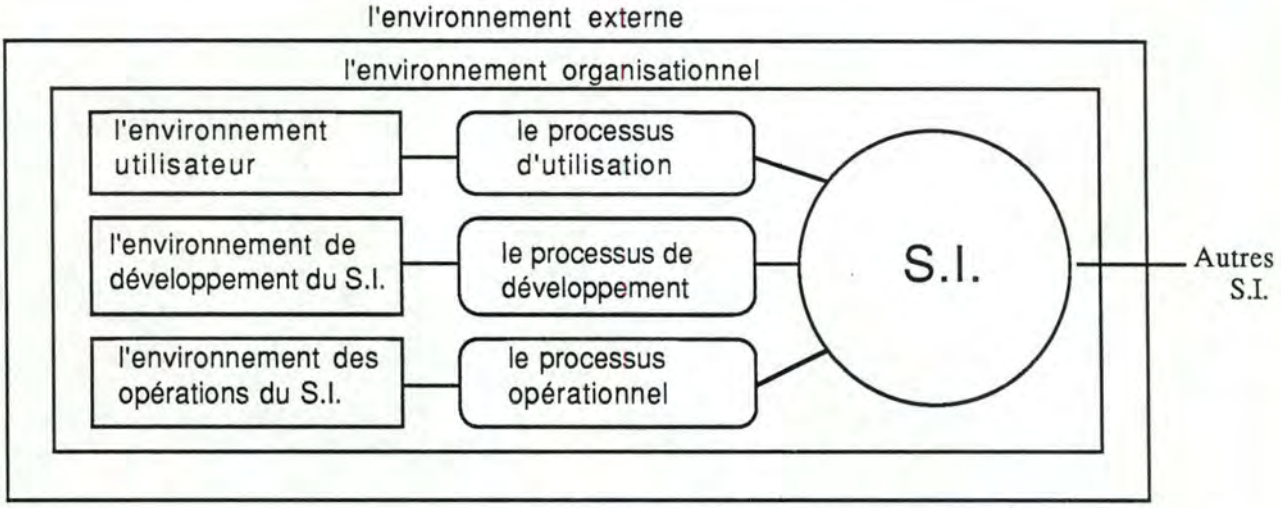


figure 1.1: Les variables infuençant le développement d'un système d'information

Les **environnements** définissent les ressources et les contraintes influençant l'étendue et la forme de chaque sous-système d'information. Cinq classes d'environnement peuvent être identifiées:

- l'*environnement extérieur* englobe toutes les considérations d'ordre légal, social, politique, culturel, économique, éducatif, commercial, etc... .

- l'*environnement organisationnel* concerne tout ce qui a trait aux objectifs, tâches, structures et style ou philosophie de gestion de l'organisation à laquelle appartient le bureau à automatiser.

- l'*environnement utilisateur* est l'environnement incluant lesutilisateurs béneficiant directement des outils informatiques installés dans le bureau. Ces utilisateurs dits "primaires" se composent exclusivement des personnes appelées à prendre des décisions dans l'organisation. Cet environnement peut être décrit grâce aux caractéristiques de l'utilisateur, de son organisation et de sa tâche.

- l'*environnement de développement du système d'information* est constitué des méthodes et techniques de développement, du personnel de conception et de ses caractéristiques, de l'organisation, de la gestion du développement et de la maintenance du système d'information. Il peut également contenir d'autres systèmes d'information pouvant communiquer avec le système en cours de développement.

- l'*environnement opérationnel du système d'information* reprend toutes les ressources nécessaires pour l'exécution du système. Cela comprend, entre autres, les logiciels, le hardware, les bases de données, l'organisation et la gestion des opérations du système d'information et les utilisateurs opérationnels (équipes techniques, équipes de gestion des bases de données ...). On remarque que ce personnel ne bénéficie pas directement, dans l'exécution de sa tâche, des services et résultats fournis par le système

d'information.

Les **processus** représentent, pour leur part, les interactions dynamiques entre le système d'information, les environnements du système d'information et les autres processus. Ives distingue ainsi:

- le *processus de développement* produit le système d'information au moyen de la sélection puis de l'application des ressources de l'organisation déterminées en tenant compte des contraintes imposées par l'environnement organisationnel. Ce processus est évalué en termes de temps et de coûts mais aussi par son impact sur l'environnement organisationnel et le niveau d'implication de celui-ci (satisfaction, participation au développement, ...).

- le *processus opérationnel* est évalué en terme d'utilisation des ressources, des performances, de la satisfaction des utilisateurs opérationnels et enfin, par les services rendus aux utilisateurs primaires.

- le *processus d'utilisation* a trait à l'utilisation du système d'information par l'utilisateur primaire et est évalué en termes de qualité de la prise de décision et de son effet sur la productivité.

Le **sous-système d'information** est le troisième facteur important du modèle. Il représente le résultat du processus de développement. On peut distinguer dans un sous-système d'information:

- Le *contenu du sous-système* qui s'exprime en termes des données et des modèles rendus disponibles pour l'utilisation du système d'information.

- La *forme de représentation* qui reprend les différentes façons de présenter l'information à l'utilisateur. Cela inclut une sélection des média (écran, imprimante, ...), du format (graphiques, tableaux, ...) et de caractéristiques spécifiques (couleurs, ...).

- La *dimension temporelle* qui peut être définie, par exemple, par le délai de traitement.

Tous ces environnements et processus jouent un rôle important dans le développement d'un système d'information mais ne sont pas exempts de problèmes. De plus, ils sont fortement liés, ce qui a tendance à renforcer et à propager ces problèmes. Par exemple, une mauvaise gestion du processus de développement, dans lequel les utilisateurs ne participent en aucune manière, aura sans aucun doute des conséquences au niveau du processus d'utilisation.

Pour notre part, nous nous concentrerons essentiellement sur les relations entre l'environnement de l'utilisateur primaire, l'environnement de développement du système d'information, le processus de développement et le système d'information. Nous nous intéresserons plus particulièrement aux problèmes de communication

entre:

- l'**utilisateur** qui est la personne dont le travail va être partiellement ou totalement automatisé par l'analyste

et

- l'**analyste** qui est la personne chargée de l'analyse et de l'automatisation proprement dite des tâches de bureau

ainsi qu'à la méthode de travail et au processus de développement appliqués par ce dernier.

S'il désire rendre son travail efficace, l'analyste procédera généralement en différentes étapes qui correspondent aux phases de déroulement du cycle de vie d'un système d'information de bureau. Ce sont ces différentes étapes que nous allons étudier dans le point suivant.

1.2 Le modèle du cycle de vie d'un système d'information

Il est classique de distinguer cinq étapes dans le cycle de vie d'un système d'information [BOPI-83] : l'étude d'opportunité, l'analyse conceptuelle, la conception de mise-en-oeuvre, la réalisation et la mise au point, et l'utilisation et maintenance (figure 1.2). Ce modèle est généralement présenté comme un enchaînement linéaire de diverses étapes avec de fréquents retours en arrière:

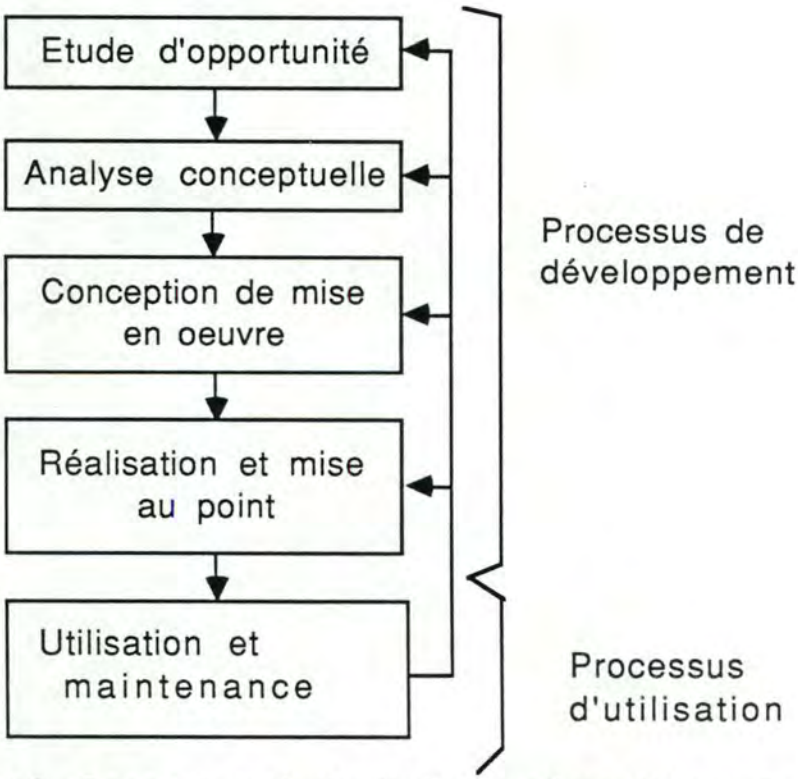


figure 1.2: Etapes du cycle de vie d'un système d'information

Une seule personne est censée concevoir le système d'information. Elle a une vue rationnelle de la situation à automatiser. A chaque étape, on peut synthétiquement

associer les tâches suivantes [LYYT-87]:

Première étape: Etude d'opportunité

Cette étape constitue une phase d'étude et de définition du bureau tel qu'il existe avant l'automatisation. La tâche de l'analyste consiste à bien comprendre le comportement du personnel afin d'obtenir une idée la plus exacte possible du travail à automatiser.

Toutefois, l'analyste ne peut pas se contenter d'une description du bureau en terme des tâches qui y sont exécutées. En effet, l'objectif d'un bureau ne consiste pas simplement à manipuler de l'information mais à poursuivre une mission (ex: négocier des contrats, effectuer des paiements...) assignée au bureau par l'organisation dont il fait partie. Finalement, les opérations élémentaires et journalières que l'on observe généralement ne servent qu' à réaliser ces différentes missions [HAMM-80].

L'étude de l'existant doit permettre de comprendre et d'identifier la fonction réalisée par le travail du bureau, l'automatisation des opérations élémentaires constituant cette mission n'étant envisagée par l'analyste que lorsque celui-ci estime que cela contribue à une meilleure réalisation de la mission.

Le résultat de cette étape est généralement un avant-projet de solution réalisé à partir d'une définition des missions assignées au bureau et des tâches existantes.

Deuxième étape: Analyse conceptuelle

Sur base de l'existant et de l'avant-projet de solution déterminés à l'étape précédente et en fonction des exigences formulées par les responsables, l'analyste peut proposer une solution possible à l'utilisateur, solution qui sera discutée et remaniée jusqu'à obtenir, généralement sur papier, le système désiré.

Troisième et quatrième étape: Conception de mise en oeuvre et réalisation et mise au point

Une fois la deuxième étape terminée, l'analyste se trouve en présence des spécifications de la solution à implémenter, solution que l'utilisateur est supposé avoir comprise et acceptée. Il peut donc passer à la réalisation pratique de son projet, ce qui suppose l'acquisition éventuelle de matériel, la réalisation de programmes, l'installation et le test des composants matériels et logiciels du système. Enfin, la réalisation comporte aussi une phase de formation des utilisateurs et l'intégration du système avec les autres opérations du bureau.

Cinquième étape: Utilisation et maintenance

Il s'agit de toutes les activités de maintenance, d'évaluation, d'amélioration et de gestion du système. Chaque nouveau changement ou amélioration à apporter implique nécessairement une nouvelle analyse des tâches existantes et une nouvelle itération du processus.

La qualité principale de cette façon de voir les choses est principalement la

capacité qu' elle attribue au concepteur de prévoir les résultats de chaque décision de conception et également, la possibilité de contrôler le processus de développement.

Cependant, très souvent, on se rend compte que les organisations n'enregistrent que des succès mitigés lors de l'implémentation de systèmes automatisés, malgré la réelle capacité que possèdent ces systèmes à résoudre des problèmes de décision, et la sophistication croissante du matériel technique et logiciel sur lequel ils sont basés.

Les principaux défauts de ce modèle du cycle de vie et de la méthode de travail qui lui est associée, sont les suivants:

- L'utilisateur est trop peu impliqué dans le développement du système et ne développe aucun sentiment de propriété à son égard. Il ne comprend généralement pas les spécifications trop techniques que lui présente le concepteur et, lors du test d'acceptation, il ne saisit pas ce qu'il faudrait modifier ni comment modifier. Ces conditions mènent, la plupart du temps, à un produit peu satisfaisant et, par le fait-même, sous-utilisé [LUCA-78].

- Cette méthode implique une certaine rigidité dans le développement. En effet, le concepteur doit s'efforcer, autant que possible, de passer par toutes les étapes pour être certain de réaliser un S.I. satisfaisant [LYYT-87].

- Le modèle insiste davantage sur les phases d'analyse et de conception et donne trop de responsabilités au concepteur. Le système lui-même est jugé trop abstrait durant les premières phases du développement. Par conséquent, l'utilisateur ne parvient à comprendre le produit qu'aux dernières étapes du cycle et donc, les occasions d'émettre des remarques et d'imposer des retours en arrière à l'analyste sont trop limitées [LUCA-78].

De plus, on remarque que l'utilisation de cette méthode se base sur une hypothèse implicite mais abusive établissant que les spécifications sont fixées à l'avance et une fois pour toutes, ce qui ne semble pas correspondre à la réalité.

Face à ces inconvénients, deux autres modèles du cycle de vie sont habituellement présentés comme alternatives: *l'approche par prototypage* et *l'approche évolutive* .

1.3 L'approche par prototypage

1.3.1 Définition

Il s'agit d'un modèle de processus rapide et peu coûteux pour développer et tester "une première version du système à réaliser qui en montre les caractéristiques essentielles" [ALAV-84]. Cette première version, que l'on appelle prototype, constitue une base concrète permettant l'évaluation des exigences supposées de l'utilisateur, l'architecture du système à venir ou la logique d'un programme. Elle montre également les propriétés et les caractéristiques essentielles du système d'information futur. Cette méthode peut être utilisée lorsque les demandes de l'utilisateur s'avèrent ambiguës et quand il est nécessaire de passer par une phase d'expérimentation et d'apprentissage

des ressources.

L'idée originale de l'approche par prototypage est que l'analyse conceptuelle et l'établissement des spécifications fonctionnelles sont développés en parallèle de leur implémentation [LYYT-87].

1.3.2 Les usages du prototypage

On accorde généralement trois usages possibles au prototypage: la clarification des exigences de l'utilisateur, la vérification de la faisabilité d'une conception et la création d'un système final [CANN-81].

- Clarification des exigences de l'utilisateur

Très souvent, l'utilisateur ne sait pas exprimer de façon précise ce qu'il désire comme système d'information. Parfois même ne sait-il pas exactement ce qu'il veut. L'usage d'un prototype doit permettre de lui éclaircir les idées. En effet, sur base d'une première discussion avec l'utilisateur, l'analyste construit un premier système réalisant quelques-unes des fonctionalités demandées, que ce soit au niveau des processus de traitement de l'information ou au niveau de l'interface utilisateur.

Suite à une utilisation de ce prototype, l'utilisateur peut vérifier si l'analyste a bien compris ce qu'il désire et peut éventuellement réclamer une modification des spécifications. Sur base de ces remarques, l'analyste améliore le prototype et le soumet de nouveau à la critique de l'utilisateur. Ce processus se poursuit jusqu'au moment où les spécifications sont parfaitement établies. Grâce à cette méthode, l'analyste comprend mieux les désirs de l'utilisateur et peut donc ainsi lui procurer un système utile.

- Vérification de la faisabilité d'une conception

Afin de tester la faisabilité des exigences de l'utilisateur, un premier prototype du système est réalisé. Ce prototype devra produire les résultats demandés mais sans aucune exigence quant à l'efficacité de l'exécution. Malgré ce manque de performances, le prototype donne, à l'analyste et à l'utilisateur, l'occasion d'utiliser le nouveau système. Sur base de leurs réactions, un deuxième prototype est développé, où le niveau d'efficacité sera un facteur important.

- Création d'un système final

Dans la version finale du prototype, obtenue grâce à la création successive de versions intermédiaires en fonction des nouvelles demandes suscitées par leurs utilisations, certaines composantes peuvent être jugées suffisantes par l'utilisateur.

En fait, le prototypage n'est utilisé que pour découvrir les véritables fonctions du système et est nécessairement suivi de la méthode classique de développement exposée précédemment (point 1.2). En effet, lorsqu'on développe des prototypes, la question d'efficacité ou plus généralement, les problèmes techniques sont souvent laissés de côté pour se concentrer uniquement sur les fonctionnalités que doit remplir

le système.

1.3.3 Avantages et inconvénients du prototypage

L'avantage de cette méthode de développement d'un système d'information est notamment d'offrir à l'utilisateur une occasion de participer à l'élaboration de son futur outil de travail. Grâce au prototype, les deux acteurs - l'analyste et le futur utilisateur - possèdent un moyen tangible pour mieux se comprendre mutuellement. Lorsque l'utilisateur est satisfait du prototype, l'analyste est assuré d'avoir à sa disposition une base sûre pour développer la totalité du système.

Un des inconvénients est que, lors de l'élaboration des spécifications, le prototypage peut conduire à un manque de soins dans l'élaboration de celles-ci. C'est pourquoi il doit s'accompagner d'une analyse rigoureuse pour éviter d'entrer dans un jeu d'essais-erreurs qui finira par lasser l'utilisateur.

1.3.4 Généralisation de l'approche par prototypage: l'approche évolutive

Cette approche, qui est une deuxième variante du modèle classique, se sert de l'approche par prototypage comme support.

Elle a pour objectif la **participation** active de l'utilisateur dans le processus de développement. En se basant sur les différentes versions qui lui sont proposées, l'utilisateur propose des idées qui doivent servir de guides au travail de l'analyste. Le système d'information continuera d'évoluer en fonction des nouvelles expériences de l'utilisateur. C'est la raison pour laquelle il n'y a pas vraiment de produit final dans un système évolutif.

1.4 Délimitation de notre travail

1.4.1 Les objectifs du travail

Automatiser un bureau est un travail beaucoup plus complexe qu'il n'y parait. Cette tâche est notamment caractérisée par des problèmes de développement du nouveau système et d'utilisation des nouveaux moyens technologiques mis en oeuvre. Cette dernière difficulté est sans doute due aux caractéristiques particulières du personnel opérant dans les bureaux, notamment le fait que les personnes dont le travail va être automatisé sont des utilisateurs peu réguliers des outils logiciels et matériels.

Parmi les nombreuses personnes qui se sont penchées sur le problème, beaucoup ont suggéré que les problèmes liés aux systèmes d'information de bureau pourraient être éliminés ou tout au moins atténués grâce à l'utilisation de meilleurs méthodes et outils de développement.

C'est pourquoi notre objectif est d'apporter à l'analyste un outil de travail qui devrait l'aider dans sa tâche d'étude et d'automatisation du bureau. L'utilisation de cet outil devrait avoir des répercussions sur sa méthode de travail et lui permettre de concilier les avantages et les inconvénients des deux modèles précédemment

exposés (points 1.2 et 1.3).

1.4.2 Le modèle du processus de développement proposé

Deux modèles de processus de développement de systèmes d'information servent de base à la conception du système d'information. Le processus classique de développement offre, comme nous l'avons vu, grâce à l'enchaînement séquentiel d'étapes bien distinctes, la possibilité de très bien contrôler l'ensemble du processus [LYYT-87].

Par ailleurs, ce processus présente de nombreux inconvénients. Le plus important d'entre eux est sans doute le manque d'implication du futur utilisateur dans les premières étapes du processus de développement.

Or les recherches dans ce domaine ont démontré l'importance d'une collaboration étroite entre l'analyste et le futur utilisateur [LYYT-87]: ces deux acteurs sont, en effet, très souvent amenés à dialoguer afin de préciser les buts et les problèmes à résoudre et de confronter les différents points de vue concernant le développement du nouveau système. Ces discussions entraînent inévitablement des problèmes de communication et de compréhension car l'analyste et l'utilisateur appartiennent à des mondes de travail distincts, ne partagent généralement pas le même langage et ont des opinions et jugements très éloignés de l'organisation dans laquelle l'automatisation doit s'insérer.

Ces difficultés de communication ont déjà été soulevées par divers auteurs et de nombreuses solutions ont été proposées [LYYT-87].

De son côté, la méthode dite par prototypage de développement de système d'information offre à l'utilisateur une occasion de participer pleinement à l'élaboration de son futur outil de travail.

Cependant, avec l'utilisation du prototype, surgit un problème supplémentaire: le manque d'outils de travail (outil de simulation, ...etc...).

L'idée de notre projet est d'utiliser le prototypage à chacune des étapes du cycle de vie (figure 1.3).

En outre, afin de faciliter la tâche de l'analyste et de lui permettre de mener à bien la construction d'un prototype dans les premières phases du processus, nous proposons l'utilisation d'un outil dont nous allons exposer les grandes caractéristiques.

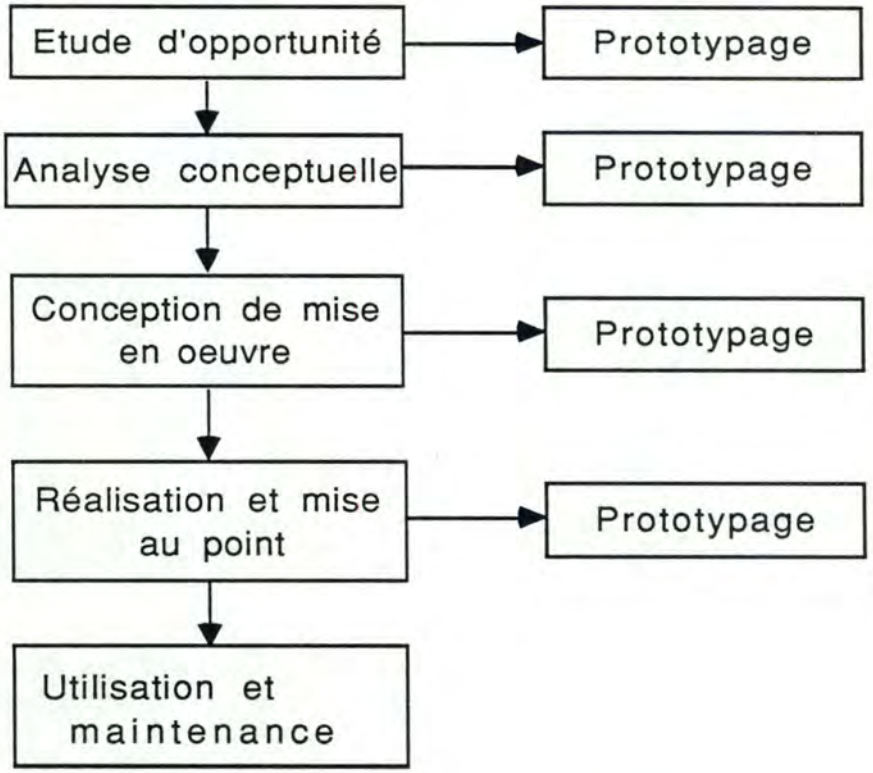


figure 1.3: Introduction de la méthode du prototypage dans le cycle de vie d'un projet.

1.4.3 L'outil proposé

L'objectif essentiel de cet outil est de faciliter la communication entre l'analyste et le futur utilisateur durant le prototypage réalisé lors de l'étape d'étude d'opportunité et de l'analyse conceptuelle. Afin de réaliser cela, son principe de base sera l'**animation graphique** à l'écran des tâches à automatiser et des solutions offertes à l'utilisateur.

Toutefois afin d'obtenir un degré d'efficacité suffisamment élévé lors de l'utilisation de l'interface graphique, l'accent devra être mis sur la qualité de ce qui sera présenté à l'écran. En effet, dans le but de ne pas perturber l'utilisateur et de susciter chez lui de bonnes réactions, ce qu'apercevra l'utilisateur devra lui paraître le plus familier possible et devra provoquer chez lui le sentiment d'être en face de quelque chose de connu, en l'occurrence son environnement de travail ainsi que les tâches qu'il effectue ou qu'il aura à effectuer. En d'autres mots, la simulation sera basée sur des *métaphores* , celles-ci étant de deux types. La première, une métaphore *fonctionnelle* [CLAN-83] fera en sorte que ce qui est présenté ressemble et produise le même comportement que ce connait déjà l'utilisateur. La seconde, une métaphore *organisationnelle* [CLAN-83], doit permettre à l'utilisateur d'associer une signification précise à chacun des objets qu'il voit à l'écran, comme il le ferait dans la réalité. Ainsi, une pile sur un bureau ne consiste pas seulement à ranger de l'information mais peut posséder une autre signification, comme par exemple lui rappeler un travail qu'il a à faire.

Voyons maintenant ce que pourrait être un tel outil et en quoi il peut s'avérer utile dans chacune des deux premières étapes du développement.

1.4.3.1 Utilisation de l'outil lors de l'étude d'opportunité

La mise au point de spécifications correctes est un problème difficile dû premièrement aux problèmes de communication entre l'analyste et l'utilisateur et deuxièmement aux ambiguités du langage naturel utilisé pour décrire ces spécifications. Ce problème se révèle d'autant plus important que toute la suite de l'analyse aura pour base les spécifications établies à cette première étape. Si celles-ci ne sont pas correctes, la suite ne le sera probablement pas non plus !

L'outil proposé devrait permettre d'améliorer la rédaction des spécifications. En effet, l'analyste aura à sa disposition un langage formalisé à l'aide duquel il définira les objets se trouvant dans le bureau analysé et la description, telle qu'il l'aura perçue, du travail effectué par l'utilisateur. Il se trouvera donc en face d'un définition aussi précise que possible du travail à automatiser. Nous pourrions schématiser cette étape de la manière présentée à la figure 1.4.

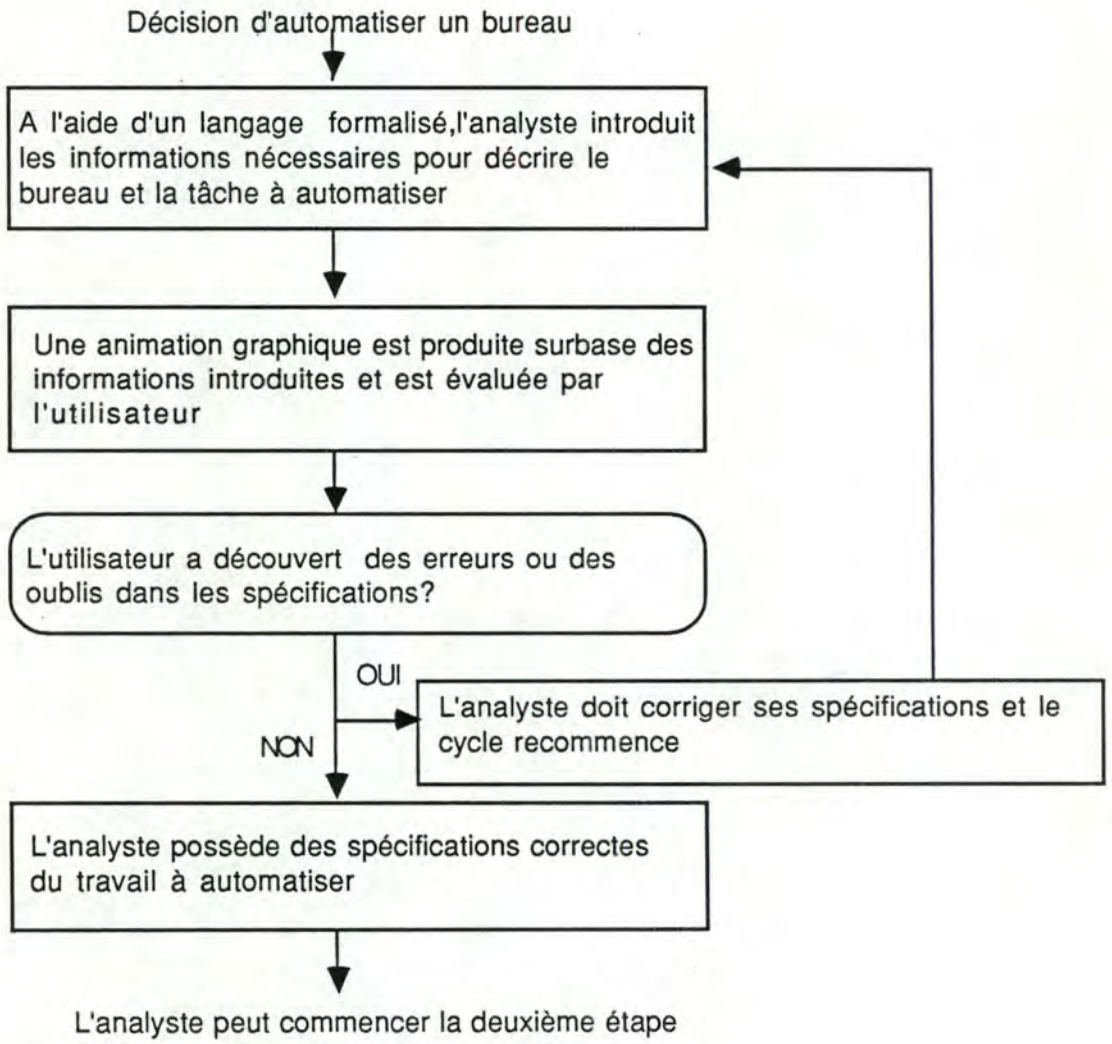


figure 1.4: Processus itératif pour la mise au point des spécifications.

1.4.3.2 Utilisation de l'outil lors de l'analyse conceptuelle

Ici encore, le problème de la compréhension entre l'analyste et l'utilisateur va jouer. Le risque est que l'analyste sera tenter de considérer, dans les explications qu'il donnera à l'utilisateur, certains aspects informatiques comme étant évidents pour lui et ne les approfondit pas.Ceci peut mener à un désintérêt des utilisateurs face au système final qui leur sera proposé si celui-ci ne produit pas du tout les résultats auxquels ils s'attendaient.

Comme dans la première étape, le projet a pour but de fournir à l'analyste un outil qui permettrait de faciliter son dialogue avec l'utilisateur, le principe de base restant encore une fois l'animation graphique. La démarche reste globalement la même que celle mise en oeuvre lors de la première étape et permet de simuler à l'écran la nouvelle implentation de la tâche proposée par l'analyste. L'utilisateur peut ainsi réagir face à cette solution, ce qui permettra à l'analyste de la corriger.

Cette étape peut se schématiser de la manière présentée à la figure 1.5:

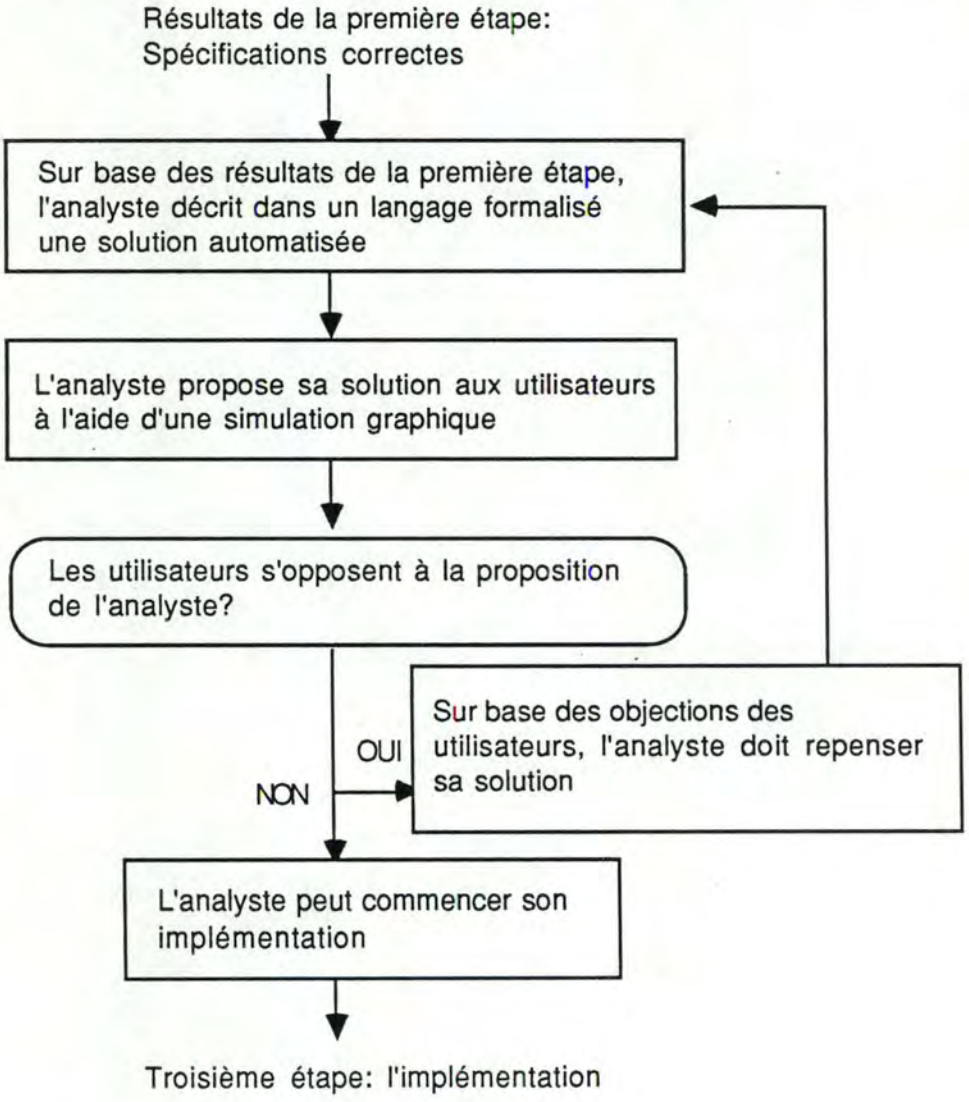


figure1.5: Processus itératif pour la mise au point d'une solution automatisée.

1.4.4 Cadre et originalité du travail

L'originalité du mémoire réside dans les trois points suivants:

- l'utilisation d'une interface graphique de haut niveau
- l'utilisation du prototypage lors de l'étude d'opportunité et de l'analyse conceptuelle
- la possibilité d'apprentissage du futur système offerte à l'utilisateur

Ces trois caractéristiques sont fortement liées. C'est pourquoi, dans ce qui va suivre, nous n'essayerons pas de les expliquer de façon distincte.

L'orignalité première de ce projet est certainement l'utilisation d'une interface de haut niveau permettant des simulations graphiques des tâches à automatiser ainsi que des solutions proposées par l'analyste. Cet outil prend place dans un domaine de recherche encore peu développé à l'heure actuelle.

Le projet de réaliser un logiciel graphique d'aide à la conception de systèmes d'information de bureau a déjà été le cadre de recherches à l'institut d'informatique [ROBE-86] [DACH-86]. Celles-ci ont notamment conduit, l'année dernière, à la réalisation d'un premier prototype [VPMS-87].

Notre objectif, dans le chapitre suivant, sera de passer en revue ces études, d'en recenser les limites afin de tenter de les lever et de réaliser un outil concrétisant les deux premières étapes du processus de développement en nous efforçant de combler les lacunes que présentait le premier prototype.

Afin de mieux cerner nos objectifs, nous allons, dans le chapitre suivant, synthétiser et évaluer les résultats des études antérieures.

CHAPITRE 2: ETUDES ANTERIEURES ET CRITIQUES

Notre travail porte sur l'élaboration d'un outil d'aide à l'automatisation du travail de bureau. Un tel outil est utile pour assister l'analyste dans son entreprise de conception d'un système automatisé et pour lui donner la faculté de modifier, selon les besoins de l'utilisateur, les spécifications du système.

Dans ce chapitre, nous passerons en revue les différents objets manipulés dans un bureau et les différentes opérations qui s'exécutent sur ces objets. Nous rappelerons les principales notions relatives au langage déterminé dans les travaux précédents et nous présenterons enfin le prototype réalisé l'an dernier, en émettant certaines critiques. Mais, dans un premier temps, examinons deux concepts importants: le bureau et la tâche de bureau.

2.1 Concepts de bureau et de tâches

Le terme "automatisation de bureau" sous-entend l'utilisation de technologies informatiques et électroniques puissantes. Mais il y a risque de ne se concentrer uniquement que sur ces aspects techniques en perdant de vue le point essentiel dans toute automatisation, c'est-à-dire le travail de bureau lui-même. C'est pourquoi, nous déterminerons dans un premier temps, ce que nous entendons par bureau, en définissant un des concepts les plus importants, à savoir la *tâche* de bureau.

Un bureau est "une unité de traitement de l'*information* dans l'*organisation* " [NEWM-80], dans laquelle "une *personne* ou un groupe de personnes exécutent des *tâches* qui consistent en un ensemble de *procédures* ." [MAPE-87]

Une tâche, pour une part, est définie comme "un travail destiné à atteindre un *but* et pour lequel une personne ou un groupe de personnes est responsable" [NEWM-80]. On peut dire également qu'une tâche est "un travail discernable dans le bureau, impliquant des *procédures* caractéristiques, des *ressources* et des *produits* . Un produit est le résultat discernable d'une tâche, qui peut servir comme mesure de la performance de ceux qui exécutent la tâche" [HINE-85]. Par exemple, le nombre de textes tapés peut être la mesure de la tâche dactylographie qui suppose des opérations de lecture, de correction ... effectuées grâce à des ressources telles que le papier, la machine à écrire ou le dactylographe.

Le développement d'un système automatisé exige donc la compréhension des tâches réalisées dans le bureau. Pour cela, il faut procéder à une analyse du bureau pour comprendre:

- les tâches
- les produits résultant de ces tâches
- les différents types de personnel impliqués dans ces tâches
- les procédures ou opérations qui permettent de réaliser les tâches et qui règlent la vie du bureau.

Il est entendu que les effets et les coûts dépensés dans l'analyse dépendront certainement de la nature et de la complexité de la tâche à automatiser. La complexité

est mesurée en terme de quantité d'information, du nombre d'opérations à effectuer et du personnel impliqué dans la réalisation de la tâche en question [HINE-85].

L'analyste détermine donc comment les tâches sont réalisées. Pour chaque tâche, il s'interroge sur

- les inputs de la tâche: qu'est-ce qui déclenche la nécessité d'exécuter une tâche? Sous quelle forme apparaissent ces inputs? ...
- la tâche elle-même: Comment est-elle réalisée? Par qui? Pourquoi? Quand?
- les outputs de la tâche: Quels en sont les résultats? Quelle forme ont-ils? ...

Les procédures, suivies par le personnel pour réaliser une tâche, sont exécutées grâce à de l'information et de la technologie qui comprend les outils de bureau et les moyens de communication. Nous allons examiner ces points plus en détail.

2.2 Les supports d'information et la technologie

Dans le cas des bureaux, il est difficile de trouver une méthodologie précise sur laquelle l'analyste pourrait s'appuyer pour analyser le système de bureau existant et concevoir un système automatisé. Il manque des modèles de l'activité de bureau. Un bon modèle devrait être simple, c'est-à-dire compréhensible par l'analyste et l'utilisateur, complet et devrait permettre la simulation de tâches quelconques [NEWM-80].

Notre travail se trouve dans la continuité d'études qui, entre autres, avaient pour but d'établir un modèle "qui soit le plus proche possible de la réalité pour qu'il puisse servir d'environnement à la simulation de tâches de bureau" [VPMS-87]. Ce modèle se base sur le modèle d'organisation de Leavitt établi dans une optique de représentation d'un bureau (figure 2.1).

Ce modèle, basé sur le modèle des organisations de Leavitt, fait apparaître les deux concepts essentiels d'un bureau:

- l'*information* manipulée par une tâche et qui est véhiculée par l'intermédiaire d'un support d'information

- la *technologie* utilisée par la tâche et qui regroupe les outils du bureau et les moyens de communications existants.

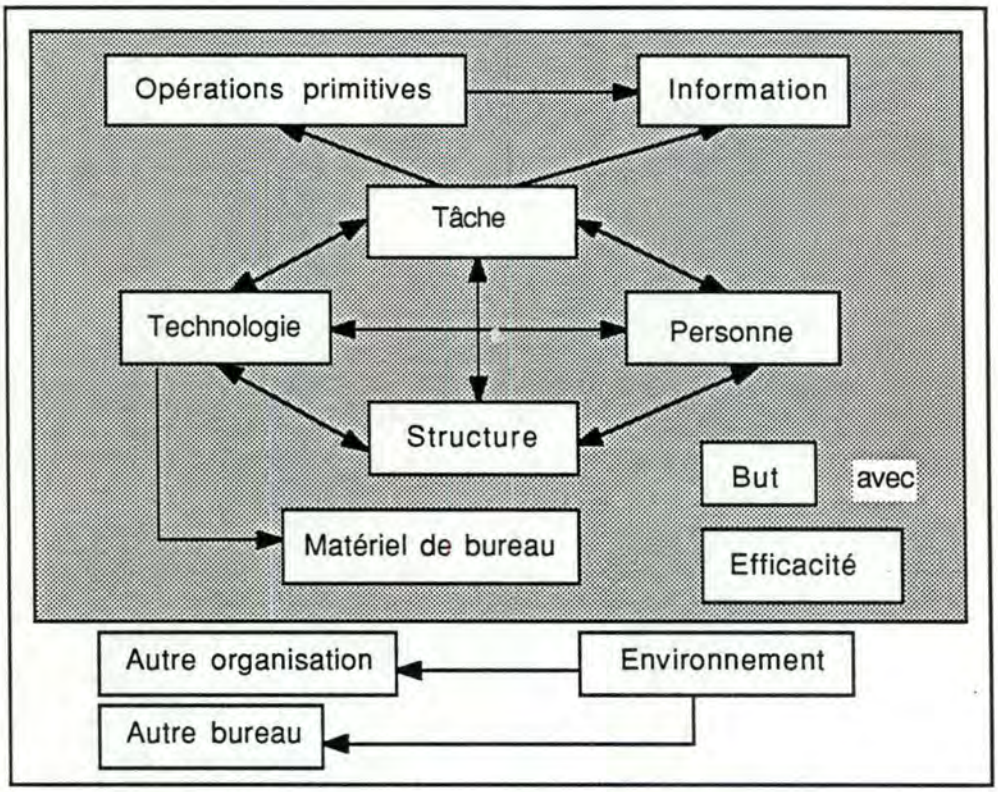


figure 2.1: Modèle des organisations de Leavitt appliqué au bureau

2.2.1 Les supports d'information

La raison d'être d'un bureau est essentiellemnet la manipulation et la communication de l'information sous toutes ses formes. Les formes d'information ont été étudiés par Dachouffe [DACH-86] qui a réalisé un modèle d'information de bureau. Il en est ressorti une série d'objets informationnels qui représentent l'information se trouvant dans un bureau et utilisée lors de l'exécution d'une tâche.

2.2.1.1 Les objets informationnels

Un objet informationnel est un objet du bureau qui est "constitué d'un support et de l'information qui s'y trouve inscrite" [VPMS-87]. Le nombre d'objets informationnels manipulés dans un bureau est très important. Parmi ces objets, on distingue deux classes: les objets informationnels élémentaires et les objets informationnels structurants. Pour chacune de ces classes d'objets, nous allons voir qu'il existe trois types différents d'objets informationnels. Les définitions sont reprises de [VPMS-87].

a. Les objets informationnels élémentaires

Ce sont les objets informationnels qui ne peuvent pas, logiquement, être décomposés pour générer d'autres objets informationnels. Parmi ces objets, on opère une classification selon la manière d'inscrire l'information sur le support:

* les *messages* sont des objets informationnels sur lesquels une information

brêve est écrite de façon non structurée et sans soins particuliers.
Ce sont, par exemple, des aides-mémoires ou des notes qui sont communiquées entre personnes d'un même bureau.

* les *formulaires* sont des objets informationnels constitués d'un certain nombre de champs libellés par un titre et pouvant recevoir une valeur.
L'information s'y inscrit lorsqu'on remplit les champs.

* les *documents* sont des objets informationnels sur lesquels est écrite une information volumineuse mais dont la mise en page est généralement structurée et soignée.
Ce sont, par exemple, des rapports contenant graphiques et tableaux, des lettres etc...

b. Les objets informationnels structurants

Il s'agit des objets informationnels constitués d'objets informationnels élémentaires et/ou en objets informationnels structurants. On peut, à tout moment, leur retirer un de leurs composants. L'information totale qu'ils contiennent se répartit dans tous les objets informationnels qui les composent.Parmi ces objets, on opère également une classification en fonction de propriétés communes (par exemple, la raison du regroupement des objets, l'ordre de classement...) :

* les *dossiers* sont des objets informationnels regroupant des informations traitant un même sujet. Par conséquent, un dossier peut être constitué d'objets informationnels de tout type, du moment que l'information qu'ils contiennent soit relative à un même sujet.
Par exemple, un dossier étudiant sera constitué de pièces très différentes telles que des formulaires d'inscription, les résultats des années précédentes...

* les *fichiers* sont des objets informationnels composés d'objets de même type, structurants ou non.
Par exemple, un fichier étudiant est composé d'un ensemble de formulaires reprenant les renseignements signalétiques d'un étudiant. Les formulaires peuvent être classés, entre autres, sur le nom de l'étudiant.

* les *piles* sont des objets infomationnels composés d'objets de tout type (à l'exception d'autres piles). Les piles servent de support à des informations sans aucun lien logique entre elles. Ces objets informationnels sont classés dans l'ordre chronologique inverse.
Par exemple, le courrier reçu est mis en pile, en attente d'un traitement.

En raison de la définition de la pile, une certaine restriction doit s'imposer quant au contenu possible des dossiers et des fichiers. Pour ces objets informationnels structurants, il existe une relation qui lie leurs composants (même sujet ou même type d'objets). C'est pourquoi, il est impossible qu'une pile en fasse partie car les constituants d'une pile n'ont aucun lien logique entre eux.

2.2.2 La technologie

Pour manipuler et communiquer de l'information, le personnel de bureau doit suivre toute une série de procédures qui sont exécutées avec l'aide de ressources qui sont soit des outils de bureau, soit des moyens de communication. Les outils de rangement ont été étudiés par Robert [ROBE-86] qui a réalisé une modélisation des objets de rangement dans un bureau . Les outils de travail et de communication résultent d'une étude effectuée par Van Pevenaeyge et Simoens [VPMS-87].

2.2.2.1 Les objets de rangement

Il s'agit des objets physiques servant de lieu de rangement aux objets informationnels. Parmi ces objets, on distingue:

* les *fardes* qui sont des objets qui ne peuvent être composés d'aucun autre objet de rangement et qui peuvent contenir n'importe quel objet informationnel.

* les *tiroirs* qui peuvent contenir des fardes et/ou n'importe quel objet informationnel.

* les *armoires* qui contiennent des tiroirs.

* les *boîtes* et les *boîtes archives* qui peuvent contenir des fardes et/ou n'importe quel objet informationnel.

* les *étagères* qui peuvent contenir des boîtes, des fardes et/ou n'importe quel objet informationnel.

2.2.2.2 Les outils de travail

Dans un bureau, il existe de la technologie qui facilite l'exécution d'une tâche. Cette technologie englobe déjà les objets de rangement vus précédemment. Il faut y ajouter les outils de travail qui permettent à l'utilisateur de traiter de l'information matérialisée par les objets informationnels. Ces outils sont principalement:

* la *table de travail* qui permet d'effectuer un travail à partir de l'information utilisée. L'information, préalablement amenéesur la table de travail, peut être consultée, vérifiée, ...
* la *photocopieuse* qui permet à l'utilisateur de dupliquer uniquement des objets informationnels élémentaires.

2.2.2.3 Les objets d'interface

Nous avons vu qu'une des principales fonctions du bureau était la communication ou la réception d'information. L'échange d'information s'effectue entre le bureau et soit un autre bureau de la même organisation, soit un bureau d'une autre organisation.

Les objets permettant ces échanges sont appelés objets d'interface et peuvent se définir comme des objets permettant soit de communiquer de l'information vers l'extérieur, soit de recevoir de l'information venant de l'extérieur. Les objets d'interface ont des caractéristiques propres qui les distinguent les uns des autres. Certains objets sont voués à la communication orale, d'autres à la communication écrite:

* le *téléphone* permet de recevoir ou d'envoyer de l'information par la voix.

* la *boite IN* est utilisée pour recevoir tout objet informationnel en provenance de l'environnement du bureau.

* la *boite OUT* est utilisée pour envoyer tout objet informationnel à destination de l'environnement du bureau.

* la *poubelle* est un objet d'interface particulier. Elle permet, en effet, la destruction de tout objet informationnel. En fait, elle sert à communiquer l'objet informationnel à détruire, à un environnement qui n'est ni un bureau, ni une organisation mais un endroit d'où l'objet sera irrécupérable.

Cette brêve description des objets, tels qu'ils ont été définis dans les travaux précédents, nous permet de découvrir les principaux éléments que l'on identifie habituellement dans un bureau. Cependant, la description des objets et de leurs caractéristiques n'échappent pas à une critique fondamentale qu'il nous semble nécessaire de signaler.

2.2.3 Critique

La critique essentielle à formuler est le caractère fermé de l'ensemble des objets présentés ci-dessus. En effet, les études antérieures offrent une classification dans laquelle n'entrent pas en ligne de compte des objets tels que le télex, les microfilms, le terminal d'ordinateur, etc ...

Par conséquent, l'analyste se trouve en face d'un appareil conceptuel trop limitatif pour parvenir à simuler le bureau de la manière la plus conforme possible en regard de la situation réelle.

2.3 Les opérations primitives

2.3.1 Introduction

Le travail de bureau est procédural par nature et implique l'exécution de *procédures* qui sont "des séquences prédéfinies d'*opérations* exécutées par une ou plusieurs personnes en vue de réaliser une tâche particulière" [NEWM-80]. Nous ne tenons pas compte ici du cadre dont le travail consiste généralement à prendre des décisions concernant l'entreprise dans laquelle il travaille.

L'automatisation du travail de bureau implique que les tâches et donc les procédures suivies dans le bureau soient correctement spécifiées. Cependant, cette spécification ne se fait pas sans difficulté, en raison de la grande variété d'activités qui caractérise le travail de bureau.

Chaque bureau est unique. Les variations entre bureaux s'expriment en terme de tâches réalisées, d'organisation du travail et de la fonction générale du bureau dans l'organisation. Il serait donc impossible de décrire le travail de bureau et encore moins de l'automatiser si l'on ne faisait pas appel à une approche fréquemment utilisée qui est le développement de *taxonomies* des tâches.

Une taxonomie des tâches est une catégorisation des activités effectuées dans un bureau. Pour établir une taxonomie, il faut observer le travail de bureau et identifier chaque tâche. Pour chaque tâche, l'analyste doit se demander qui l'exécute, la durée de chaque exécution, si la tâche nécéssite ou non une communication, si c'est le cas, quelle modalité est suivie pour communiquer, quel est le but de chaque activité, etc ... La taxonomie permet de repérer les tâches de bureau les plus importantes sur lesquelles on concentrera les efforts d'amélioration technologique. Les taxonomies représentent une base commune permettant à l'analyste et à l'utilisateur de se comprendre.

Malgré le fait qu'il n'existe pas de bureau semblable à un autre, on peut identifier des similitudes entre les tâches exécutées dans des bureaux différents. Tous les bureaux sont des centres de communication. Ce sont des lieux où l'information est reçue et envoyée aussi bien que traitée.

En général, on peut identifier quatre types de tâches:

* les tâches de communication orale ou écrite d'information

* les tâches de traitement des informations

* le classement / rangement d'information

* la création de l'information

Chaque tâche implique l'exécution de procédures qui sont une séquence d'opérations dites primitives s'exécutant d'abord sur les objets informationnels (2.2.1.1) mais mettant en oeuvre également les objets de rangement (2.2.2.1), les outils de travail (2.2.2.2) et les objets d'interface (2.2.2.3). Voici un bref rappel des

opérations primitives définies lors des travaux antérieurs (figure 2.2 et 2.3).

2.3.2 Critiques

De la même façon que nous avons critiqué les objets de bureau, nous estimons que la liste des opérations primitives présentée ci-dessous est loin d'être complète. Comme nous l'avons déjà fait remarquer, les bureaux se différencient les uns des autres en raison de la variété importante des activités que l'on peut y observer. C'est pourquoi l'analyste doit pouvoir simuler n'importe quelle tâche de bureau. Si l'on ne dispose que des opérations primitives définies aux figures 2.2 et 2.3, la simulation ne pourra que refléter partiellement la situation réelle du bureau.

Opérations primitives	Objet informationnel élémentaire		
	message	*formulaire*	*document*
Création			
- Créer	X	X	X
Communication orale ou écrite			
- Recevoir	X	X	X
- Communiquer	X	X	X
- Détruire	X	X	X
Classement/Rangement			
- Archiver		X	X
- Classer	X	X	X
- Ranger	X	X	X
- Replacer	X	X	X
Traitement			
- consulter	X	X	X
- modifier			X
- completer		X	
- décider		X	X
- feuilleter			
- enlever			
- vérifier		X	
- trier			
- reproduire	X	X	X

figure 2.2: Liste des opérations primitives associées aux objets informationnels élémentaires

Opérations primitives	Objet informationnel structurant		
	dossier	*fichier*	*pile*
Création			
- Créer	X	X	X
Communication orale ou écrite			
- Recevoir	X	X	
- Communiquer	X	X	
- Détruire	X	X	X
Classement/Rangement			
- Archiver	X	X	
- Classer	X	X	
- Ranger	X	X	X
- Replacer	X	X	X
Traitement			
- consulter			
- modifier			
- completer			
- décider	X		
- feuilleter	X	X	X
- enlever			X
- vérifier	X		
- trier	X	X	X
- reproduire			

figure 2.3: Liste des opérations primitives associées aux objets informationnels structurants.

2.4 Le langage de description d'environnement et de tâches de bureau

2.4.1 Introduction

Comme cela a déjà été dit (cfr chapitre 1), un analyste a besoin d'un langage formalisé pour définir les tâches qu'il désire simuler à l'écran .

Notre intention n'est pas ici d'exposer complètement ce langage mais plutôt d'en donner les caractéristiques générales. Les lecteurs interessés pourront se référer au mémoire de Van Pevenaeyge et Simoens [VPMS-87] qui ont créé ce langage. Nous donnerons d'abord un aperçu des conventions adoptées pour la description syntaxique du langage et nous passerons ensuite au langage de description d'environnement et de tâches. Enfin, nous montrerons en quoi le langage ainsi défini peut être sujet à la critique.

2.4.2 Les conventions syntaxiques

Les conventions suivantes seront utilisées pour décrire la syntaxe des phrases du langage, une phrase pouvant se définir comme une suite de mots permettant de décrire un opération primitive [VPMS-87]:

- les mots en majuscule sont des mots réservés du langage
- les mots en minuscule sont des mots dont les valeurs sont choisies par l'utilisateur
- les mots entourés de crochets **[ ]** sont des clauses optionnelles à l'intérieur d'une construction
- les mots entourés de parenthèses **( )** sont des clauses répétitives
- les mots entourés d'accolades **{ }** et disposés en colonnes ne peuvent être utilisés simultanément mais au moins une des options doit être utilisée.

2.4.3 Le langage de description d'un environnement de bureau

Avant de définir et/ou de simuler une tâche, l'analyste doit avoir spécifié et créé un environnement de bureau dans lequel l'exécution de la tâche est sensée commencer.

Le texte correspondant à la définition de cet environnement est constituée d'une suite de **phrases** où chaque phrase décrit la création d'un objet du bureau ou d'une association entre deux objets distincts. De plus chacune des phrases doit répondre à la syntaxe développée dans [VPMS-87].

2.4.3.1 La déclaration des objets

Grâce au langage, il est permis de créer des objets informationnels, des objets de rangement, des objets de travail et des objets d'interface. A chacune de ces classes d'objets correspond une phrase de création spécifique. Lors de la définition d'un objet, un certain nombre de données servant à qualifier l'objet devront être spécifiées. Ainsi par exemple, lors de la création d'un document, devront être obligatoirement précisés le nom et le type du document ainsi qu'éventuellement l'état

et/ou l'échéance et/ou le numéro de copie. Ces informations qualitatives sont appelées des *attributs* .

A titre d'exemple, nous reprenons ci-dessous la description syntaxique de la phrase utilisée dans le cas de la création d'objets informationnels:

```
CREER OBJET INFORMATIONNEL
   { [COPIE NUMERO n DE] MESSAGE nom-message }
   { [COPIE NUMERO n DE] FORMULAIRE DE TYPE type-formulaire
                         IDENTIFIE PAR nom-formulaire }
   { [COPIE NUMERO n DE] DOCUMENT DE TYPE type-document
                         IDENTIFIE PAR nom-document }
   { DOSSIER DE TYPE type-dossier IDENTIFIE PAR nom-dossier }
   { FICHIER nom-fichier }
   { PILE nom-pile }
[AVEC ETAT =  nom-etat] [AVEC ECHEANCE = date]
[AVEC ORDRE DE CLASSEMENT = { ALPHABETIQUE }
                            { ALPHABETIQUE INVERSE }
                            { FIFO }
                            { LIFO }
                            { AUCUN }]
```

Une application de ceci pourrait être la création d'un fichier de nom "étudiants", d'état "créé" et d'ordre de classement "LIFO":

```
CREER OBJET INFORMATIONNEL FICHIER étudiants AVEC ETAT = créé
   AVEC ORDRE DE CLASSEMENT= LIFO
```

2.4.3.2 La déclaration des associations entre les objets

Après avoir spécifié les objets qui appartiennent au bureau, l'analyste doit définir les associations existant entre eux. Le terme association désigne le concept du même nom présent dans le modèle Entité-Association de [CHEN-76] qui fut d'ailleurs utilisé par [VPMS-87] pour modéliser les objets de bureau. Une association est définie par Bodart et Pigneur [BODA-83] de la manière suivante: "une association est définie par une correspondance entre deux ou plusieurs entités (non nécessairement distinctes, où chacune assure un rôle donné.". Ces associations peuvent être de quatre types possibles en fonction des objets manipulés:

- le type d'association **classement** qui représente le classement d'un objet informationnel élémentaire ou structurant dans un objet informationnel structurant.

- le type d'association **rangement** qui représente le rangement d'un objet informationnel élémentaire ou structurant dans un objet de rangement.

- le type d'association **traitement** qui représente le traitement d'un objet informationnel élémentaire sur un outil de travail.

- le type d'association **interface** qui représente la communication d'un objet

informationnel de ou vers le bureau.

- le type d'association **composition** qui représente la hiérarchisation des objets de rangement.

Comme dans le cas de la création des objets, à chaque type d'association correspond une phrase type dont la syntaxe à respecter est également reprise dans [VPMS-87]. Ici aussi, des informations devront être données afin d'identifier valablement les objets à associer.

On pourrait citer comme exemple, la phrase de création d'une association de rangement entre un objet informationnel et un objet de rangement:

CREER ASSOCIATION RANGEMENT ENTRE
{ [COPIE NUMERO n DE] MESSAGE nom-message }
{ [COPIE NUMERO n DE] FORMULAIRE DE TYPE type-formulaire
IDENTIFIE PAR nom-formulaire }
{ [COPIE NUMERO n DE] DOCUMENT DE TYPE type-document
IDENTIFIE PAR nom-document }
{ DOSSIER DE TYPE type-dossier IDENTIFIE PAR nom-dossier }
{ FICHIER nom-fichier }
{ PILE nom-pile }
ET { ETAGERE nom-etagere }
{ BOITE nom-boite }
{ TIROIR nom-tiroir }
{ FARDE nom-farde }

La création d'une association de rangement entre un dossier de nom "étudiant1" de type "inscription" et l'étagère de nom "étagère-dossier" pourrait donc s'écrire:

CREER ASSOCIATION RANGEMENT ENTRE DOSSIER DE TYPE inscription IDENTIFIE PAR étudiant1 ET ETAGERE étagère-dossier.

2.4.4 Le langage de description de tâches de bureau

Une fois la composition du bureau établie, l'analyste peut procéder à la rédaction du texte de description d'une tâche devant se dérouler dans le bureau. Dans ce but, l'outil utilisé consiste également en un langage de haut niveau qui pour être utile et agréable à manipuler, doit permettre d'une part de refléter le mieux possible la définition générale d'une tâche de bureau et d'autre part doit être suffisamment souple de manière à ne pas imposer trop de contraintes à l'utilisateur dans la rédaction de son texte.

Une tâche peut être considérée comme un enchaînement d'opérations primitives qui manipulent de l'information (point 2.3). Dès lors, afin de bien refléter cette définition, l'outil devra offrir à l'analyste:

- la possibilité de spécifier, dans un formalisme bien précis, chacune des opérations intervenant dans la description de la tâche

et

- des structures d'enchaînement permettant d'établir un lien entre chacune des opérations primitives qui composent la tâche. Ces structures seront la séquence, la boucle et la condition.

Quant à l'objectif de souplesse de manipulation, il peut être raisonnablement atteint par l'utilisation d'un langage non procédural qui permet à l'analyste de déclarer ses opérations primitives et ses structures d'enchaînement dans un ordre quelconque.

2.4.4.1 Les principes de base du langage

La description d'une tâche est un texte constitué d'une suite de **phrases** du langage où chaque phrase représente la description d'un **composant.** Tout composant devant être d'un des cinq types suivants:

- le type tâche
- le type sous-schéma
- le type condition
- le type boucle
- le type opération primitive

De plus entre les différents composants d'une tâche, on trouvera des relations de type **déclenche** qui permettront de définir l'ordre dans lequel ceux-ci doivent être exécutés. Ces relations peuvent être schématisées de la manière suivante:

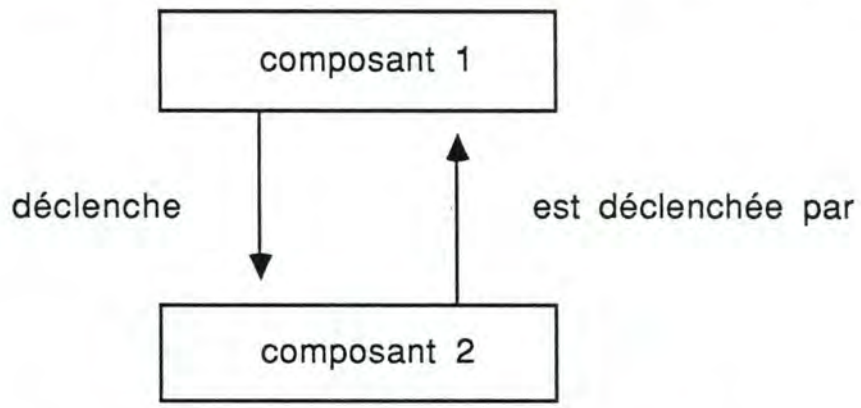


figure 2.4: Schématisation des relations de déclenchement entre deux composants

Ceci signifie que:

- chaque composant de la tâche doit être décrit dans une seule phrase
- tout composant décrit dans une tâche doit avoir un nom qui l'identifie parmi tous les composants de la tâche.

A chacun des types de composant cités ci-dessus correspond une phrase spécifique du langage. La description syntaxique de ces phrases ainsi qu'une représentation graphique des composants peuvent être trouvées dans [VPMS–87], nous nous contenterons pour notre part de reprendre la définition de chacun de ces correspondants.

2.4.4.2 Le composant tâche

La définition du concept de tâche a déjà été donnée au point 2.1; quant au composant tâche, on peut dire que son rôle se limite à rassembler l'ensemble des composants de la tâche sous un même nom: celui de la tâche. Ces composants forment ce qu'on appelle un sous-schéma. La seule chose que puisse déclencher un composant de type tâche est son sous-schéma. Regrouper les composants de la tâche sous un même nom sera utile losqu'il pourra y avoir une communication décrites dans le système. On pourrait imaginer la simulation d'une application de bureau qui se décomposent en un enchaînement de tâches qui, elles-même, seraient un enchaînement de composants.

Voyons maintenant ce qu'est un sous-schéma.

2.4.4.3 Le composant sous-schéma

La notion de sous-schéma peut se définir comme étant un enchaînement de composants qui ont un lien logique entre eux. Ainsi, les composants qui se trouvent dans le corps d'une boucle ont un lien logique entre eux: ils font partie du corps de la même boucle. En cela, nous pouvons dire que le corps d'une boucle constitue un sous-schéma. De la même manière, tous les composants d'une tâche ont un lien logique entre eux: ils font tous partie de la même tâche. Nous pouvons dès lors également affirmer que tous les composants d'une tâche forment un sous-schéma.

Dans une certaine mesure, on peut comparer cette notion de sous-schéma à celle de bloc utilisée dans les langages tels que PASCAL ou C.

En ce qui concerne les composants déclenchés par un sous-schéma, ils peuvent être de n'importe quel type, à l'exception du type tâche. De plus, la fin de chaque sous-schéma devra être marquée par un composant spécial appelé *fin du sous-schéma* .

2.4.4.4 Le composant opération primitive

La liste des opérations primitives applicables à un objet informationnel est reprise au point 2.3. Pour insérer une opération primitive dans la description d'une tâche de bureau, l'analyste devra la déclarer dans un composant de type opération primitive à l'aide du formalisme décrit dans [VPMS-87].

Ce formalisme met en oeuvre les notions de paramètre et de variable. Nous allons tout d'abord examiner rapidement ce que sont ces deux concepts.

- la notion de paramètre

Un paramètre possède un nom qui l'identifie parmi tous les noms de paramètres qui existent dans le système. Il désigne un emplacement de la mémoire destiné à recevoir une réponse de l'utilisateur à l'écran. Un paramètre peut recevoir une valeur soit lors de l'exécution d'une opération de *vérification*, soit lors de l'exécution d'une opération de *décision* .

- la notion de variable

Une variable porte un nom qui l'identifie dans l'ensemble des variables du système et contient des informations (classe, nom, numéro de copie, type) permettant d'identifier un objet informationnel appartenant à un bureau.

Grâce à ce mécanisme, l'analyste peut manipuler des objets informationnels sans devoir connaître les informations identifiantes, l'assignation d'un identifiant à une variable se faisant à l'aide soit d'une *boucle* soit de l'opération primitive *enlever*.

Les seuls types de composants pouvant être déclenchés par un composant de type opération primitive sont les composants boucle et condition. L'introduction dans son texte d'une phrase décrivant un composant d'un de ces types, permettra à l'analyste de créer un enchaînement autre que simplement séquentiel entre les opérations primitives.

A titre d'exemple, nous citerons la phrase décrivant l'opération primitive VERIFIER:

VERIFIER { [COPIE NUMERO n DE] FORMULAIRE DE TYPE type-formulaire
IDENTIFIE PAR nom-formulaire }
{ DOSSIER DE TYPE type-dossier IDENTIFIE PAR nom-dossier}
{ VARIABLE nom-variable }
[AVEC ETAT = nom-etat] [AVEC ECHEANCE = date]
[AVEC RESULTAT VERIFICATION nom-parametre]

La réponse que l'utilisateur introduira, affectera le paramètre dont le nom est spécifié. Quant à la variable identifiée par nom-variable, elle contiendra l'identifiant d'un dossier ou d'un formulaire.

2.4.4.5 Le composant condition

Le composant condition peut être exprimée de la manière suivante: si une certaine condition est vérifiée, alors tel composant est exécuté, sinon tel autre l'est. Une condition peut être formée de trois manières:

1) tester la valeur d'un attribut d'un objet informationnel spécifié directement ou référencé par l'intermédiaire d'une variable.

2) tester l'appartenance d'un objet informationnel, spécifié ou référencé par une variable, à un objet structurant donné qui peut lui-même être référencé par une variable.

3) tester la valeur d'un paramètre.

Les composants susceptibles d'être déclenchés par une condition sont les composants opérations primitives, boucle et condition.

2.4.4.6 Le composant boucle

Ce type de composant permet à l'analyste de sélectionner un ensemble d'objets informationnels sur base d'un certain critère et d'appliquer le même traitement - décrit dans un ensemble de composants - à chacun des objets contenus dans cet ensemble.

Les critères de sélection sont:

1) sélection de
   - soit tous les objets informationnels élémentaires,
   - soit tous les objets informationnels structurants,
   - soit tous les objets informationnels constituant un objet informationnel structurant.

2) sélection de tous les objets informationnels d'une même classe.

3) sélection de tous les objets informationnels du bureau.

Le traitement à réaliser sur chacun des objets sera décrit dans un sous-schéma. L'exécution de ce type de composant est alors la seule à pouvoir être déclenchée par la boucle et elle le sera pour chaque objet sélectionné. Après avoir épuisé l'ensemble des objets, la boucle peut déclencher l'exécution de n'importe quel type de composant à l'exception d'une tâche.

Tel est sommairement le langage formalisé utilisé par l'analyste pour décrire le bureau à automatiser ainsi que les tâches qui s'y déroulent. Le lecteur intéressé trouvera, en annexe 1, un exemple d'application des concepts que nous venons de passer en revue.

2.4.5 La critique du langage

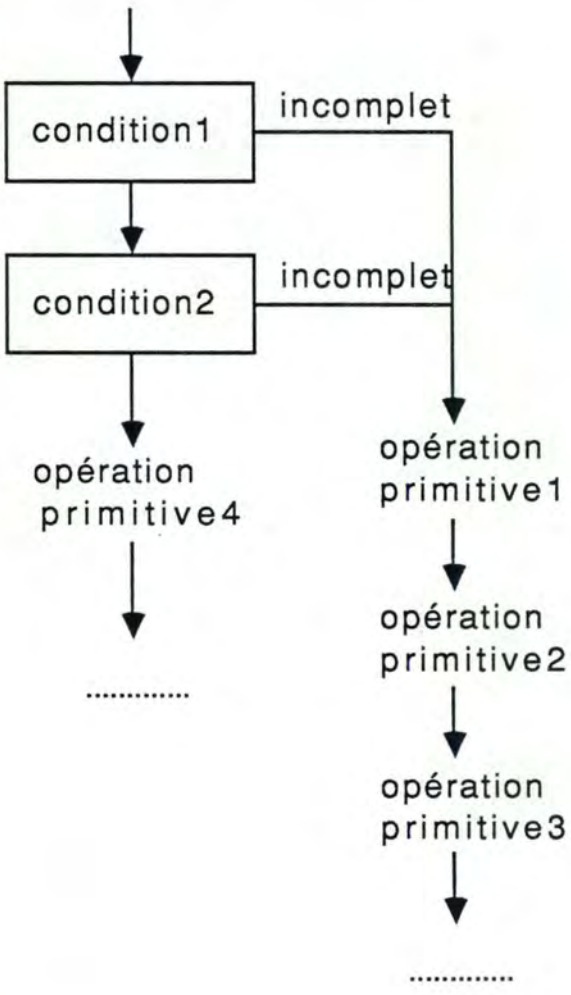
Le langage tel qu'il vient d'être décrit apparaît très agréable à utiliser pour l'analyste parce que d'une part, il reste finalement assez proche du langage naturel et que d'autre part, grâce à son caractère non procédural, il permet de définir les composants d'une tâche dans un ordre quelconque. De plus, cet aspect non procédural du langage offre un autre avantage non négligeable qui est la suppression de la redondance dans la rédaction du texte décrivant une tâche. En effet, supposons que dans un langage plus "classique" où la notion de composant telle que nous l'avons défini n'existe pas, on ait décrit l'enchaînement d'opérations primitives suivant:

```
.....
si paramètre1 = "incomplet"
   alors: [opération primitive1
           opération primitive2
           opération primitive3
           ..... ]
   sinon: [si paramètre2 = "complet"
              alors: [opération-primitive1
                      opération-primitive2
```

opération-primitive3
.....]
sinon: [opération-primitive4
.....]]

On observe que le texte décrivant la suite des opérations primitives 1,2 et 3 a dû être dupliqué à deux endroits différents. Etant donné la longueur relativement importante d'une phrase de description d'opération primitive, si cela devait se représenter plusieurs fois et pour différentes parties du texte, celui-ci pourrait rapidement devenir assez lourd à manipuler que ce soit lors de la rédaction ou que ce soit lors de la lecture.

De manière équivalente à la notion de procédure des langages classiques, le concept de composant permet donc de simplifier fortement l'écriture du texte en définissant une et une seule fois chacune des opérations primitives dans un composant opération primitive et l'enchaînement des diverses opérations par une série de déclenchements entre ces composants.



En utilisant cette propriété du langage formalisé proposé, nous pourrons écrire:

CONDITION condition 1
DESCRIPTION: on teste la valeur du paramètre paramètre1
ENONCE: SI PARAMETRE paramètre1 = "INCOMPLET"
DECLENCHEE PAR .....
SI VRAI DECLENCHE OPERATION PRIMITIVE opération-primitive1
SI FAUX DECLENCHE CONDITION condition2

CONDITION condition2
DESCRIPTION: on teste la valeur du paramètre paramètre2
ENONCE: SI PARAMETRE paramètre2 = "INCOMPLET"
DECLENCHEE PAR .....
SI VRAI DECLENCHE OPERATION PRIMITIVE opération-primitive1
SI FAUX DECLENCHE OPERATION PRIMITIVE opération-primitive4

OPERATION PRIMITIVE opération-primitive1
DESCRIPTION: .....
ENONCE: .....
DECLENCHEE PAR CONDITION condition1
CONDITION condition2
DECLENCHE OPERATION PRIMITIVE opération-primitive2

OPERATION PRIMITIVE opération-primitive2
DESCRIPTION: .....
ENONCE: .....
DECLENCHEE PAR OPERATION PRIMITIVE1
DECLENCHE OPERATION PRIMITIVE opération-primitive3

OPERATION PRIMITIVE opération-primitive3
DESCRIPTION: .....
ENONCE: .....
DECLENCHEE PAR OPERATION PRIMITIVE2
DECLENCHE .....

OPERATION PRIMITIVE opération-primitive4
DESCRIPTION: .....
ENONCE: .....
DECLENCHEE PAR CONDITION condition2
DECLENCHE .....

On se rend rapidement compte que les redondances dans le texte ont disparu et qu'une telle description de tâches peut s'avérer nettement plus claire.

Cependant, malgré ses aspects positifs, le langage n'en présente pas moins des inconvénients. Ses faiblesses découlent en fait des critiques que nous avons émises lors de l'aperçu des études concernant les objets (point 2.2) et les opérations primitives (point 2.3). En effet, en raison de l'optique de prototypage choisie et des outils utilisés, on n'avait pu étudier qu'un ensemble **fermé** d'objets et d'opérations primitives. Il était donc tout à fait impossible d'envisager, à l'aide de l'outil développé,

l'analyse d'un bureau contenant d'autres objets que ceux prévus. Ce problème se répercute directement sur la syntaxe même du langage qui ne tient pas compte de l'ajoût éventuel d'une nouvelle opération primitive ou d'un nouvel objet. Ce langage ne connait, en effet, qu'un nombre limité d'opérations réalisables et il impossible de "lui en apprendre de nouvelles"; de plus les quelques opérations connues ne sont utilisables que sur certains objets bien déterminés et ne le sont évidemment pas sur de nouveaux. Un exemple permet d'illustrer ce point de vue. Prenons l'opération de consultation d'un objet informationnel dont la syntaxe est la suivante:

```
CONSULTER { [COPIE NUMERO n DE] MESSAGE nom-message }
          { [COPIE NUMERO n DE] FORMULAIRE DE TYPE
                                type-formulaire
                                IDENTIFIE PAR nom-formulaire }
          { [COPIE NUMERO n DE] FORMULAIRE DE TYPE
                                type-formulaire
                                IDENTIFIE PAR nom-formulaire }
          { VARIABLE nom-variable }
[AVEC ETAT = nom-etat] [AVEC ECHEANCE = date]
```

On s'apeçoit immédiatement que l'ajout d'un nouvel objet informationnel élémentaire - que l'on voudrait évidemment pouvoir consulter - est rendu impossible par suite de la rigidité de la syntaxe. De même, la consultation éventuelle d'un objet informationnel structurant n'est pas non plus envisageable. Dès lors, si l'on veut rendre le logiciel applicable à des cas réels, le langage doit être adapté.

Outre cette contrainte gênante, certaines autres limites moins flagrantes apparaissent également au moment de l'utilisation du langage. A titre d'exemple, on peut citer entre autres:

- l'impossibilité de dupliquer facilement un objet informationnel **structurant** (pile, dossier, fichier) à l'aide d'une opération primitive, si ce n'est par duplication de chacun de ses composants.

- l'incapacité de ranger quelque chose dans la table de travail, ce qui semble assez pauvre pour respecter la réalité.

- l'obligation lors de la création d'une boîte in ou d'une boîte out, de créer un objet pile et de l'associer explicitement à la boîte. La création de la pile et de son association pourrait tout aussi bien se faire automatiquement lors de la création de toute boîte in ou out.

2.5 La faisabilité du projet

Les résultats des recherches tels que nous les avons exposés, bien qu'extrêmement interessants, ne prouvent malheureusement en aucune manière qu'un projet aussi ambitieux que celui dans lequel nous sommes impliqués, puisse être mené à bien. En effet, tout ce que nous avons écrit jusqu'à présent ne peut être finalement considéré que comme un ensemble d'idées cohérentes mais qu'il reste à pouvoir mettre en oeuvre.

Si l'on se réfère à ce que nous avons écrit à propos de la méthode de développement de systèmes d'information par prototypage et de ses usages (point 1.3), on s'aperçoit que l'utilisation d'un prototype peut permettre dans certaines situations, de vérifier la faisabilité d'une conception sans aucun souci de performance ou d'efficacité.

C'est en quelque sorte, cette démarche qui fut appliquée précédemment par Van Pevenaeyge et Simoens afin de démontrer la faisabilité de l'outil proposé dans le cadre de ce projet.

2.5.1 Le prototype

Le prototype réalisé est en réalité le premier produit développé dans le cadre d'une démarche évolutive de développement d'un outil graphique d'aide à la conception de systèmes d'information de bureau, et ne peut donc pas être considéré comme une version finale du système. Son seul objectif n'était d'ailleurs pas de fournir une simulation de haut niveau des tâches décrites par l'analyste, mais plutôt de montrer qu'un projet tel que celui défini était tout à fait concevable.

Le prototype développé permettait de répondre aux exigences formulées, en ce sens qu'il offrait la possibilité dans une certaine mesure de définir un environnement de bureau et des tâches qui y sont exécutées. Les informations introduites sont contrôlées et stockées dans une base de données de style tout à fait classique et servent de matières premières à une simulation graphique sur écran, les données étant mises à jour en fonction des modifications apportées aux objets du bureau suite à l'exécution d'opérations primitives.

Finalement, on peut affirmer qu'il remplit bien sa mission puisqu'effectivement, il montre que l'idée de construction d'un tel outil est entièrement envisageable.

Cependant, comme tout produit résultant des premières étapes d'une démarche de développement par ajustements progressifs, ce prototype présente de nombreux points faibles et peut donc être soumis à la critique.

2.5.2 Critique du prototype

2.5.2.1 Critique du modèle

Les inconvénients majeurs du prototype développé sont classés par ordre d'importance décroissante, les trois premiers demandant à notre avis le maximum d'attention si l'on désire rendre le système effectivement applicable à des situations réelles

a) Le nombre limité d'objets disponibles

L'implémentation effectuée du programme de prototypage ne prenait pas en compte l'existence, dans les bureaux, d'objets autres que ceux qui furent définis et se limitaient à un petit ensemble très - et même trop - fermé. L'outil actuel est donc tout à fait inadéquat lors d'applications courantes où l'on pourrait très bien se trouver confronté à un problème d'automatisation d'un bureau où seraient présents des objets

de type machine à écrire, terminal, télex, etc... .

Si l'on désire employer cet outil de manière efficace, il est donc absolument indispensable de lui adjoindre un mécanisme d'ajoût de nouveaux types d'objet. Ce mécanisme devra se montrer suffisamment puissant en vue de réaliser rapidement l'ajoût d'un nouveau type d'objet et de permettre l'application de n'importe quelle opération primitive à un objet du type ainsi créé, mais il devra également être agréable et aisé à manipuler.

b) Le nombre limité d'opérations primitives

Comme c'est le cas pour les objets, la liste des opérations primitives prise en compte lors de l'implémentation précédente, est beaucoup trop figée et est loin de couvrir l'ensemble de ce qui est effectivement exécuté dans la réalité. Rien ne nous empêche d'imaginer qu'un analyste puisse avoir besoin de définir une tâche dans laquelle apparaîtrait une opération absente de la liste qui lui a été fournie. Dans l'état actuel des choses, si le cas se présentait, l'outil ne pourrait donc lui être d'aucune aide.

Encore une fois, comme dans le cas des objets, un système d'ajout d'opérations primitives est une nécessité. Ces nouvelles opérations pourraient être applicables aussi bien à des objets appartenant à l'ensemble prédéfini qu'à de nouveaux objets créés par l'analyste.

2.5.2.2 Critique de l'implémentation graphique

a) La définition de l'environnement de bureau

Si l'on désire que le problème de la mauvaise compréhension entre l'analyste et l'utilisateur soit résolu de manière optimale à l'aide de la simulation graphique, il faut que ce qui soit montré à l'écran soit une représentation fidèle de ce qui se passe effectivement dans le bureau analysé, sous peine d'imposer à l'utilisateur des efforts d'imagination inutiles et déplaisants.

En particulier, l'environnement de bureau présenté doit refléter le plus précisément possible l'univers dans lequel l'utilisateur est plongé quotidiennement. Or cela n'est pas rendu possible par le prototype qui, quel que soit le type et le nombre des objets déclarés par l'analyste, présente toujours invariablement la même image figée d'un bureau type, ne montre pas tous les objets du bureau et ne permet pas de les placer là où l'utilisateur le désire, selon la disposition de son propre bureau.

b) L'identification des objets

Dans l'implémentation du prototype, si deux objets d'apparence identique mais de nom et/ou de caractéristiques différentes apparaissent à l'écran, il est tout à fait impossible pour l'analyste et l'utilisateur de les distinguer.

Ce problème s'amplifie encore lors de l'exécution et de la simulation d'une tâche. Là également, aucune possibilité n'est offerte à l'utilisateur du système de savoir exactement quels objets sont manipulés; d'où un risque évident de manque de

compréhension de ce qui se déroule à l'écran. Dès lors, un des objectifs de l'outil, qui était l'amélioration de la communication entre l'analyste et l'utilisateur, n'est pas atteint.

C'est pourquoi, une identification la plus précise possible de tout objet présenté à l'écran est exigée. Cette identification pourrait se faire par affichage sur chaque objet d'une étiquette contenant son nom, ou si on dispose d'une souris, par affichage à l'écran du nom de l'objet sur lequel on pointe ou on clique.

c) Des inconvénients divers

A côté de ces inconvénients importants, le programme de prototypage réalisé présente encore un certain nombre d'autres inconvénients que l'on pourrait considérer comme secondaires en ce sens qu'ils sont plutôt d'ordre graphique et esthétique, et ne constituent en rien un obstacle à la réalisation du système. On peut citer:

- la table de travail, qui jusqu'ici, était exclusivement considérée comme un outil de travail pourrait voir ses tiroirs servir d'objets de rangement. Elle deviendrait alors un outil de travail **et** un objet de rangement, et cumulerait ainsi les propriétés de ces deux types d'objets

- quel que soit le nombre de tiroirs composant une armoire, le prototype présente toujours une armoire à trois tiroirs. Encore une fois, dans le but d'offrir une vision du bureau qui soit une image fidèle de celui de l'utilisateur, il conviendrait de lever cette limitation

- dans le même registre, une représentation des objets plus fine et à l'échelle s'avérerait plus commode. Ainsi par exemple, un téléphone n'aurait pas la même taille qu'une étagère.

Ceci constitue une série d'exemples de petits défauts du prototype. En réalité, ils pourraient être considérés comme des détails car même s'ils jouent un rôle non négligeable dans la réussite future du projet, ils ne remettent pas fondamentalement en cause sa faisabilité.

Toutefois, nous ne pouvons terminer cette critique sans rappeler que si l'on parvenait à résoudre les deux premiers problèmes cités ci-dessus, on devrait alors nécessairement repenser en conséquence le langage de haut niveau employé pour la description des tâches et de l'environnement de bureau. En effet, le manque de souplesse de sa syntaxe entraine une incapacité évidente à prendre en compte de nouveaux objets et de nouvelles opérations primitives créés par l'analyste.

2.5.3 Les justifications des défauts du prototype

Lorsqu'on se propose de développer un produit, en prenant comme référence une première version déjà implémentée, et que l'on tente d'en combler les lacunes principales, il est primordial de rechercher tout d'abord le pourquoi de ces lacunes, la démarche qui fut suivie dans la réalisation, les erreurs qui ont été commises, et de déterminer ensuite d'autres voies qui devraient nous conduire à de meilleurs résultats.

C'est ce à quoi nous allons nous livrer dans les paragraphes qui vont suivre. Trois grandes raisons expliquent, selon nous, les inconvénients et imperfections que nous avons relevées:

- l'optique de prototypage choisie par les concepteurs
- la méthode de représentation formelle des objets et de leurs associations
- les outils de programmation choisis.

2.5.3.1 La démarche par prototypage

Rappelons encore une fois que l'intention des premiers implémenteurs n'était pas de mettre sur pied un système tout à fait complet et très performant mais plutôt de sélectionner parmi les exigences des initiateurs du projet un ensemble de fonctionnalités de base et de tenter de les implémenter sur machine afin de voir si les travaux de recherche entrepris jusqu'alors étaient réalistes. Dès lors, certaines fonctionnalités furent laissées de côté, ce fut le cas entre autres de l'ajout d'objets et d'opérations primitives, ainsi que d'une définition sophistiquée de l'environnement de bureau. Encore auraient ils voulu les prendre en compte qu'ils n'auraient pu le faire faute d'outils adéquats.

2.5.3.2 La méthode de représentation formelle des objets et de leurs associations

Une cause probable des inconvénients présentés par le précédent programme de prototypage, résulte du choix du modèle conceptuel de structuration des informations. En effet, après avoir analysé le bureau et recensé la liste des objets ainsi que leurs associations, il était nécessaire d'en donner une représentation formelle à l'aide d'un modèle. Ce dernier devait dans un premier temps, permettre de dégager la signification potentielle attachée par l'utilisateur aux données qui représentent les concepts, les objets et leurs associations. On se trouve donc à un niveau conceptuel car l'objectif est seulement d'exprimer la sémantique à l'exclusion des problèmes de représentation physique des données (codage interne), d'accès aux données et d'organisation de stockage. Néanmoins, ce dernier point ne peut pas laisser indifférent. C'est ainsi que le modèle devait de plus être structuré pour rendre possible une translation directe du schéma conceptuel dans les structures de données physiques définies par un système de gestion de base de données (niveau représentatif).

Il existe un grand nombre de modèles conceptuels de structuration des informations réunissant ces deux critères. On opta pour le modèle Entité-Association de Chen [CHEN-76] en raison notamment de sa bonne capacité de représentation des informations appartenant au réel perçu et de ses grandes qualités de communication.

Cependant la réalisation du modèle entité-association d'un système d'information pose l'hypothèse que l'on a préalablement fixé de manière précise les informations ou les objets manipulés ainsi que les relations ou les associations existant entre eux. Le modèle a ainsi tendance à figer la représentation structurelle du réel et à ne pas laisser la porte ouverte aux changements éventuels provenant du monde extérieur. Il ne prend donc pas en compte la création possible d'un nouveau type d'objet ou d'une nouvelle opération primitive. Il est dès lors interessant de se

tourner vers des méthodes de modélisation et des langages de programmation disposant de mécanismes permettant de faire face à ce problème.

2.5.3.3 Les outils utilisés

Que ce soit au niveau langage de programmation, logiciel ou matériel, les outils, mis à la disposition des étudiants chargés de la conception du prototype, étaient loin d'être à la hauteur de leurs ambitions.

D'une part, le programme de prototypage précédent a été implanté au moyen du langage C et de. Ces deux choix ne sont pas sans inconvénients: il semble en effet indiquer que MS-Windows n'a pas du tout été conçu pour de l'animation graphique pure mais plutôt pour une bonne gestion de fenêtres. De même, la représentation des objets à l'aide d'icônes proposées par ce logiciel ne semble pas non plus satisfaisante.

D'autre part, le langage C utilisé n'offrait pas à un programme les fonctionnalités nécessaires de manière à augmenter les capacités de réponse face à des changements en provenance de l'environnement que les autres langages classiques tels que PASCAL, COBOL... etc... .

2.6 Nos objectifs

La conclusion de la plupart des critiques que nous avons formulées dans ce chapitre à l'égard de tous les travaux précédents, est que la réalisation d'un système tel que celui que nous envisageons nécessite une attitude différente envers le changement qui n'est pas fournie par les outils de programmation, la méthodologie et les concepts utilisés puisqu'ils n'acceptent pas notamment l'ajout de nouveaux objets et de nouvelles opérations primitives par l'analyste.

Il existe différentes écoles de pensées qui proposent des stratégies à suivre pour répondre à ces problèmes. Celles-ci sont rapidement passées en revue par Cox [BCOX-86]. Cependant l'une d'entre elles semble s'imposer de plus en plus dans de nombreux domaines de l'informatique tels que l'intelligence artificielle ou la conception assistée par ordinateur: la **programmation orientée objet**.

C'est également cette solution que nous avons adoptée afin de construire une deuxième version du programme de prototypage sur base des recherches effectuées par nos prédécesseurs. A l'aide de la programmation orientée objet et de l'environnement SMALLTALK-80, notre objectif sera de mettre en oeuvre notamment:

- un mécanisme d'ajout de nouveaux types d'objet
- un mécanisme d'ajout de nouvelles opérations primitives
- un mécanisme de définition d'environnement de bureau plus performant grâce à un déplacement possible des objets à l'aide d'une souris
- un mécanisme plus précis d'identification des objets apparaissant à l'écran
- une représentation plus fine des objets afin de mieux représenter la réalité

Quant au langage formalisé, nous tenterons de montrer comment sa définition

devrait être revue et comment son implémentation pourrait être envisagée dans un environnement orienté objet.

Mais avant de passer à la réalisation concrète de notre projet, le chapitre 3 présentera la programmation orientée objet, ses concepts et ses principes de base. Nous montrerons en quoi également elle diffère de la programmation classique et en quoi elle peut nous aider efficacement dans notre entreprise. Enfin, nous décrirons comment seront implémentés ces concepts et principes dans le langage et l'environnement que nous avons choisi, c'est-à-dire SMALLTALK-80.

CHAPITRE 3

LA PROGRAMMATION ORIENTEE OBJET

CHAPITRE 3: LA PROGRAMMATION ORIENTEE OBJET

Ce chapitre sera entièrement consacré à la programmation orientée objet. Nous commencerons par un exposé des problèmes rencontrés dans le monde du génie logiciel. Ensuite nous présenterons les principes de base mis en oeuvre en programmation orientée objet et destinés à résoudre ces problèmes, nous verrons de la sorte pourquoi ce genre de programmation a tendance à s'imposer. Après avoir procédé à un rapide survol des différents types de langages existant à l'heure actuelle, nous passerons à un examen détaillé de Smalltalk-80. Enfin, nous décrirons la méthode de travail propre au développement de programmes orientés objet et nous montrerons en quoi la programmation orientée objet peut se montrer utile dans notre mémoire.

3.1 La crise du logiciel

A la lecture de certaines statistiques [BCOX-86], on peut se rendre compte qu'encore très souvent, par rapport au nombre total de projets commencés, la proportion de programmes corrects et terminés est toujours très faible. La plupart des problèmes cruciaux que présentent généralement la majorité des logiciels sont [MEYE-87a]: l'**extensibilité**, la **réutilisabilité**, la **correction** et la **fiabilité**.

3.1.1 L'extensibilité

Le concept d'extensibilité peut être défini comme étant "la facilité avec laquelle des produits logiciels peuvent être adaptés aux changements des spécifications". [MEYE-87a]. Généralement, ce problème de résistance face aux changements ne pose pas beaucoup de difficultés avec de petits programmes; il devient par contre souvent critique lorsque ceux-ci atteignent une taille importante, un changement étant alors susceptible d'avoir des retombées à de multiples endroits dans le texte d'un programme.

La notion d'extensibilité est une question cruciale lors de la conception de tout logiciel, et dans le cas du nôtre en particulier, puisque nous avons dit combien il était sensible au changement (point 2.5.2). En effet, nous devrons, par exemple, à tout moment être capables d'ajouter un nouveau type d'objet à manipuler ou une nouvelle opération primitive à simuler de manière à rendre notre outil utilisable en toutes circonstances.

Afin d'illustrer le problème posé par l'extensibilité du logiciel, prenons un exemple de situation à laquelle nous devrons faire face dans notre travail.

Imaginons que l'on ait modélisé et implémenté physiquement un objet de type boîte qui puisse contenir n'importe quel type d'objet informationnel (point 2.2.1.1). Supposons que l'on doive concevoir un module donnant un résumé des attributs (point 2.4.3.1) qualifiant chacun des objets rangés dans la boîte. Si l'affichage des informations associées à un objet dépend des caractéristiques particulières de cet objet et que l'on doit faire appel à un module spécifique, la solution classique est

d'associer un type à chaque objet et de laisser au programme la responsabilité de choisir le module correct suivant ce type [BCOX-86]. Un module global valable pour chacun des objets connus pourrait être écrite dans un pseudo-langage du type PASCAL de la manière suivante:

```
AfficherAttributsDe (unObjet)
   case (unObjet.type) of
      FORMULAIRE: AfficherAttributsFormulaireDe (unObjet);
      DOCUMENT: AfficherAttributsDocumentDe (unObjet);
      MESSAGE: AfficherAttributsMessageDe (unObjet);
      PILE: AfficherAttributsPileDe (unObjet);
      DOSSIER: AfficherAttributsDossierDe (unObjet);
      FICHIER:  AfficherAttributsFichierDe (unObjet)
   end;
```

L'utilisation d'une structure alternative, telle que celle utilisée, implique automatiquement que le module ne s'exécutera correctement que pour des objets de type formulaire, document, message, pile, dossier et fichier. Un changement dans les spécifications, consistant par exemple en l'ajout d'un nouveau type d'objet informationnel, soit le type livre, résultera en la localisation et la modification de **tous** les tests de ce genre afin d'assurer la correction de ce module ainsi que de tous ceux bâtis sur le même schéma. Dans un système quelque peu important, ces changements peuvent vite s'avérer fastidieux et mener à de multiples erreurs.

3.1.2 La réutilisabilité

La réutilisabilité est "la capacité des produits logiciels à être réutilisés en partie ou totalement, pour de nouvelles applications" [MEYE-87a]. On observe, en effet, que de nombreux éléments de systèmes logiciels ont les mêmes fonctionnalités et de plus suivent le même schéma général. Il devrait être possible d'exploiter ces aspects communs afin d'éviter de repartir de zéro et de réinventer de nouvelles solutions chaque fois que l'on aborde un problème déjà rencontré. Il est en effet étonnant de voir le nombre de fois que dans un département informatique, on écrit des fragments de programmes qui réalisent le même genre d'opération.

L'exemple choisi au point 3.1.1 nous permet également d'illustrer certains problèmes liés à la réutilisabilité. En effet, dans le cas de l'affichage des attributs d'un objet informationnel, on s'aperçoit que chaque fois qu'un nouveau type d'objet est créé, on doit programmer un nouveau module d'affichage d'attributs qui lui est associé. Immédiatement, se fait sentir le besoin d'un module tout à fait général qui soit applicable quel que soit le type de l'objet et donc réutilisable pour de nouveaux types d'objets.

Toutefois, de nombreux problèmes vont se poser. Le premier d'entre eux est que ce module devra être applicable à n'importe quel type d'objet (dossier, pile, etc...). Or, à chacun de ces types risque d'être associée une structure de données particulière à laquelle correspondra généralement des algorithmes de travail distincts. Pour être véritablement flexible et réutilisable, un module tel que celui que l'on propose, devrait être utilisable par un programme sans que ce dernier ne connaisse ni le type réel de l'objet manipulé ni la façon dont le module est implémenté. Si, suivant le type de

l'objet, des fonctions différentes doivent être activées, un mécanisme sous-jacent, inconnu du programme, devra s'en charger.

3.1.3 La correction et la fiabilité

Si l'on désire obtenir vraiment de bons systèmes, il est absolument nécessaire de se préoccuper des notions de correction et de fiabilité. La première est la capacité d'un produit logiciel à exécuter parfaitement les tâches qui lui sont assignées et telles qu'elles ont été spécifiées. La seconde est la capacité des logiciels à fonctionner même dans des conditions anormales.

3.2 Une solution: la programmation orientée objet

La programmation orientée objet est généralement présentée comme la solution aux problèmes posés par le style et les langages de programmation traditionnels et dont certains ont été exposés ci-dessus. Elle propose, en effet, quelques nouveaux concepts et principes de programmation qui peuvent aider à la production de logiciels plus flexibles et plus extensibles.

Avant de passer à la définition détaillée des concepts liés à la programmation orientée objet, il nous a paru interessant de connaître les grands principes liés à ce type de programmation [MEYE-87a]. Chacun des points qui vont suivre met en évidence un principe que doit mettre en oeuvre tout langage pour être qualifié de langage orienté objet.

Dans l'exposé de ces principes, nous avons volontairement introduit des concepts sans les définir au préalable: nous désirons en effet, dans un premier temps, donner au lecteur les idées directrices qui ont conduit au développement de l'orientation objet et des langages qui en dérivent Ainsi par exemple, nous utiliserons le terme "entité" ou "objet" pour désigner n'importe quel objet de l'univers qui nous entoure, qu'il soit matériel (une farde, une armoire, une boîte, ...) ou immatériel (un entier, un réel, ...), que l'on pourrait désirer modéliser afin de lui appliquer des opérations.

Dans un second point, nous définirons plus formellement et de manière précise l'ensemble des concepts de base utilisés dans la conception orientée objet.

3.2.1 Les grands principes mis en oeuvre en programmation orientée objet

Premier principe:

Structure modulaire basée objet

"Les systèmes sont modularisés sur base des structures de données qu'ils manipulent plutôt que sur base des fonctions qu'ils exécutent." [MEYE-87a].

Comme nous l'avons vu précédemment, tout logiciel est appelé à évoluer dans le temps. Chaque changement apporté au système provoque la création d'une nouvelle version, elle-même susceptible de subir une nouvelle modification. Ainsi, la dernière version est, le plus souvent, totalement différente de ce à quoi on pouvait s'attendre au vu des spécifications. Cependant, il est à noter que les catégories d'objets sur lesquelles agit le système, sont toujours sensiblement les mêmes car elles ne sont en général pas touchées par l'évolution du logiciel au contraire de ses fonctionnalités. Ainsi, par exemple, un système d'exploitation utilise continuellement des mémoires, des unités de calcul, etc ... C'est pourquoi, il est préférable de décomposer le système sur base des catégories d'objets qu'il devra manipuler plutôt que sur les fonctions qu'il doit exécuter, à la condition que ces catégories soient considérées à un niveau suffisamment haut d'abstraction.

Second principe:

Abstraction des données

Toute modification des structures de données provoque le remaniement des programmes pour qu'ils s'adaptent aux nouvelles structures (point 3.1.1). Ceci montre clairement la nécessité de séparer les programmes de la structure physique des entités (ou objets) qu'ils manipulent. Ce problème se résoud grâce au principe de l'**abstraction** qui permet de "séparer la description fonctionnelle d'un objet des détails de sa réalisation" [KRAK-85]. Chaque entité doit donc être vue de l'extérieur, sans égards à sa représentation interne. Ceci sera réalisé de la manière suivante: l'état d'une entité est contenu dans des variables privées visibles uniquement via des opérations spécifiques qui leur sont associées et rassemblées dans ce qu'on appelle l'interface ou protocole de l'entité. En outre, l'utilisation d'un tel interface peut permettre de changer les structures des données internes et les procédures sans affecter l'implémentation du reste du logiciel.

Afin d'illustrer ce second principe, prenons un exemple tiré de [COLN-86]. Dans un système orienté objet, nous pourrions avoir défini un objet ARTICLE-DU-STOCK, qui, comme son nom l'indique, décrit un article en stock.

L'état d'un tel objet serait contenu dans l'ensemble des variables privées suivantes:

- REFERENCE, qui désigne le numéro référençant l'article.
- DESIGNATION, qui est un texte décrivant l'article.
- PRIX-UNITAIRE, qui indique le prix unitaire hors taxes de l'article.
- QUANTITE, qui donne le nombre d'articles du type décrit disponibles dans le stock.

Le contenu de ces variables ne pourrait être consulté ou manipulé que par l'intermédiaire des opérations appartenant à l'interface de l'objet. Ces opérations pourraient être:

- PRIX-TTC, qui calcule le prix toutes taxes comprises d'un article du stock, en ajoutant le montant de la TVA au prix hors taxes donné par la variable PRIX-UNITAIRE.
- RETIRER (COMBIEN), qui retire des articles du stock en soustrayant à la variable QUANTITE la valeur de son paramètre COMBIEN.
- AJOUTER (COMBIEN), qui ajoute des articles au stock en ajoutant à la variable QUANTITE la valeur de son paramètre COMBIEN.

Cette caractéristique est fondamentale. Elle garantit, en effet, la fiabilité et la modifiabilité ou extensibilité d'un logiciel (point 3.1) en réduisant les interdépendances entre ses composants.

Troisième principe:

Les classes

Le principe d'abstraction conduit également à regrouper, sous une seule description, des entités partageant des propriétés communes. Ces propriétés caractérisent une **classe** dont les entités sont des représentants. Cette idée a conduit à la notion de **type abstrait** : "un type abstrait de données décrit une classe d'objets à travers leurs propriétés externes et les opérations que l'on peut y appliquer plutôt que par leur représentation interne." [MEYE-87a].

Une classe représente donc l'abstraction et ses instances, des exemplaires du type décrit par la classe. Créer une classe, c'est spécifier une représentation d'un type abstrait de données qui décrira la structure de donnée et les opérations associées. L'idée est de cacher, à l'extérieur, les détails d'implémentation pour montrer uniquement les spécifications du type abstrait de données.

Si l'on reprend l'exemple du second principe, nous aurions la classe ARTICLE-DU-STOCK. Cette classe devra fournir la description des variables privées de tout article en stock. Elle devra également décrire comment les opérations appartenant au protocole d'un objet article en stock doivent être exécutées.

Quatrième principe:

L'héritage

"Une classe peut être définie comme l'extension ou la restriction d'une autre." [MEYE-87a].

Une classe B définie comme **héritière** d'une classe A reçoit toutes les propriétés

de A auxquelles B peut ajouter ses propres caractéristiques. La classe B est considérée comme un cas particulier de A car tout objet de type B peut être interprêté comme un objet de type A. Entre B et A existe une relation **sous-classe de**: B est la sous-classe de A

Le concept d'héritage est fondamental car il permet au programmeur de définir un nouveau type en ajoutant de nouvelles propriétés à un type existant, créant ainsi un **sous-type** du type existant.

Par exemple, on pourrait avoir la classe Rangement qui rassemblerait les caractéristiques générales de tous les objets de rangement d'un bureau. Une de ses sous-classes possibles serait la classe Etagere qui reprendrait les propriétés particulières à cette sorte d'objet de rangement. On dit alors que Rangement est la super-classe de Etagere. Ceci peut se représenter graphiquement de la manière suivante:

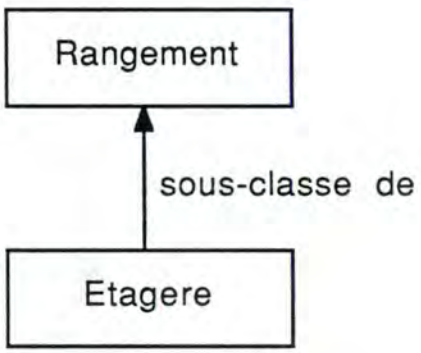


figure 3.1: Exemple de hiérarchie simple

L'utilisation de ce principe résoud de manière significative le problème de la réutilisabilité évoqué au point 3.1.2 .

Cinquième principe:

Le polymorphisme

Les principes d'abstraction et d'héritage ont conduit à définir deux caractéristiques pour les objets:

- la **généricité**, c'est-à-dire "la possibilité de paramétrer un objet (module, procédure, type, ...), les paramètres pouvant eux-mêmes être des types." [KRAK-85].

  Exemple: pile (x) est générique et peut engendrer pile (entier), pile (chaîne), ...

- le **polymorphisme**, c'est-à-dire "la possibilité pour une opération d'accepter des paramètres appartenant à différents types." [KRAK-85].

Exemple: imprimer (x) où x est de type entier, réel, ...

Grâce à ces deux caractéristiques, les programmes peuvent se référer, lors de l'exécution, à des entités appartenant à d'autres classes, et les opérations peuvent avoir différentes implémentations possibles dans différentes classes.

Le système utilisera le mécanisme de **liaison dynamique** ou **dynamic binding** pour sélectionner l'implémentation de l'opération adaptée à l'objet. Cette propriété permet de résoudre le problème de l'extensibilité tel qu'il a été rencontré au point 3.1.1.

Dans l'approche objet, contrairement à l'approche procédurale classique, ce sont les objets eux-mêmes qui sont responsables de la sélection de l'opération adéquate. La même opération peut être appliquée à des objets différents mais peut provoquer des réponses différentes selon leur classe.

Sixième principe:

L'héritage multiple

Ce mécanisme permet de déclarer une classe comme héritière de plus d'une classe et d'augmenter les possibilités de réutilisabilité.

La description de la nouvelle classe d'objets est créée à partir de la combinaison des descriptions d'autres classes. La classe hérite alors de toutes les propriétés de ces classes.

On pourrait ainsi envisager d'avoir deux classes appelées Rangement et Travail qui regroupent respectivement les caractéristiques de tous les objets de rangement et de tous les les objets de travail.

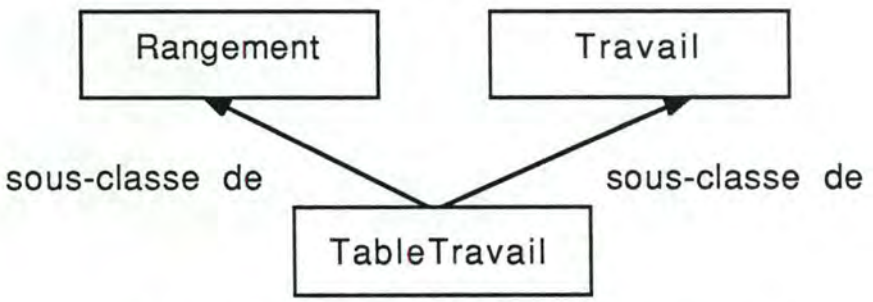


figure 3.2: Exemple de hiérarchie multiple

On pourrait alors créer une classe TableTravail qui regouperait des objets héritant à la fois des propriétés des objets de Rangement et de Travail. Cette hiérarchie est exposée à la figure 3.2.

Ces six principes font apparaître la logique suivie dans le développement la programmation par objet. Voyons à présent les concepts sous-jacents à ce nouveau style de programmation.

3.2.2 Les concepts de base de la programmation orientée objet

Les concepts que nous allons définir sont généraux à la quasi-totalité des langages et systèmes qui se rattachent à la programmation orientée objet mais peuvent apparaître avec des variantes diverses selon les cas.

3.2.2.1 Le concept d'objet

L'univers est constitué d'entités appelées **objets** . "Un objet est caractérisé par un état et comporte un ensemble d'opérations qui permettent de consulter et de modifier cet état, et d'interagir avec les autres objets." [KRAK-85].

L'objet est donc l'élément principal sur lequel se base la programmation orientée objet. Il est constitué d'une partie de mémoire privée définissant son état et d'un ensemble d'opérations dont la nature dépend du type de composant auquel elles sont associées. Ce concept rassemble les notions de données et de procédures des langages procéduraux et met en oeuvre le principe des types abstraits de données. L'ensemble de ces opérations constitue la seule possibilité pour modifier ou consulter sa mémoire privée de l'objet.

Un objet pourrait, par exemple, représenter un dictionnaire dont l'information propre serait l'ensemble de ses clés et de ses valeurs associées. Son interface serait constituée des opérations d'ajout ou des opérations d'accès aux différentes valeurs.

3.2.2.2 Les concepts de classe et d'instance

La plupart des systèmes orientés objet font une distinction entre la description d'un objet et l'objet lui-même. Beaucoup d'objets similaires peuvent être décrits par une même description générale, ce que l'on appelle une **classe** . "La description d'un objet est appelée une classe, si celle-ci peut décrire tout un ensemble d'objets associés entre eux." [ROBS-78].

Tout objet appartient à une classe et chaque objet décrit par une classe, est appelé une **instance** ou un représentant de cette classe. Les instances d'une même classe partagent des attributs communs (procédures, structures des informations d'état, ...) mais chaque instance contient également, dans des **variables d'instance** , les informations qui permettent de la distinguer des autres représentants de la classe. Toutes les instances d'une classe ont le même nombre de variables d'instance mais les valeurs de ces variables sont différentes d'instance à instance.

A titre d'exemple, on pourrait avoir la classe Formulaire qui servirait de modèle pour la création des objets qui la représentent. Elle serait caractérisée par les variables d'instance *nom*, *type* et *échéance* et les opérations *rangerDans* et

classerDans. On pourrait alors créer, par exemple, deux instances de cette classe, soit formulaire1 et formulaire2 dont les valeurs des variables d'instance seraient différentes. De manière graphique, cela peut se représenter de la manière représentée à la figure 3.3 .

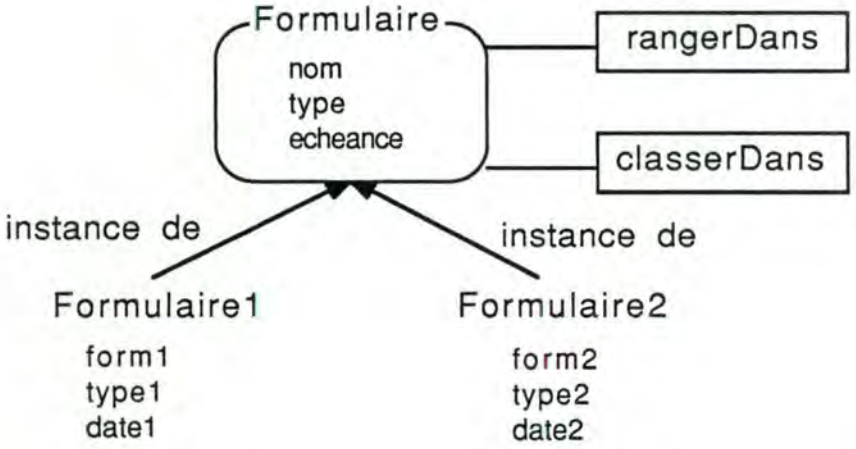


figure 3.3: Exemple d'une classe et de ses instances

A partir d'une classe, on peut par le mécanisme d'héritage, créer une sous-classe La nouvelle classe ainsi créée est une **sous-classe** de la classe existante qui est appelée la **super-classe** de la nouvelle classe. La sous-classe hérite des variables d'instances des attributs et des méthodes de sa super-classe.

La sous-classe peut en outre ajouter à sa définition, ses propres attributs ou méthodes davantage appropriés pour des objets plus spécialisés.

L'héritage peut donc permettre aux programmeurs de créer de nouvelles classes d'objets en spécifiant uniquement les différences entre la nouvelle classe et une classe existante plutôt que de repartir de zéro à chaque fois. Du code peut donc être ainsi réutilisé.

3.2.2.3 Les concepts de message et de méthode

Toutes les actions, en programmation par objet, sont déclenchées par l'envoi de messages. Au lieu d'appeler une procédure pour exécuter une opération sur un objet, un **message** est envoyé à l'objet.

Les objets communiquent entre eux par envoi de messages. Un message est "une information émise par un objet, au cours de l'exécution d'une procédure, à destination d'un autre objet; il provoque l'exécution d'une procédure par l'objet destinataire, appelé le receveur ." [KRAK-85].

Tout message comporte un nom symbolique, le **sélecteur** , qui décrit le type d'opération désirée par l'objet émetteur et non pas comment cela sera réalisé, ce qui est le cas en programmation procédurale. Quand un message est envoyé, le récepteur détermine lui-même quelle **méthode** exécuter sur base du sélecteur du message; par méthode, nous conviendrons d'entendre "la description d'une séquence d'actions

destinées à être exécutées par un processeur." [ROBS-78] et décrivant une opération.

Il est à noter que l'opération à exécuter et les paramètres de l'opération sont déterminés de façon dynamique en fonction du contenu du message. En effet, quand un objet reçoit un message, il détermine lui-même comment y répondre, c'est-à-dire qu'il sélectionne, dans son ensemble d'opérations, celle qu'il faut exécuter.

Un message peut être interprêté de différentes façons par différents récepteurs, en vertu du principe de polymorphisme (point 3.2.1).

L'ensemble des messages, auquel un objet peut répondre, est appelé le **protocole** ou **interface** de l'objet et constitue, en fait, la partie de l'objet qui est seule visible de l'extérieur.

3.3 Les langages orientés objet

Pour supporter pleinement la programmation orientée objet et donc, pour mériter le titre de langage orienté objet, tout langage doit présenter les principes définis au point 3.2.1.

Notre intention n'est pas de dresser ici une liste exhaustive de ces langages rattachés à la famille des langages orientés objets mais plutôt de souligner l'apport de l'approche par objet dans de nombreux domaines, du génie logiciel à l'intelligence artificielle. Nous nous limiterons donc à un bref survol des langages les plus représentatifs de cette catégorie. Il est à noter que chaque langage peut apparaître dans des catégories différentes.

- Le langage d'origine

  Destiné aux traitements des problèmes de simulation, Simula n'est pas à proprement parler un langage orienté objet. Mais il fut le premier à introduire les notions de classe et d'objet et le concept de généricité [RENT-82].

- Les langages apparentés

  Ce ne sont pas des langages orientés objet à part entière, mais ils permettent un style de programmation se rapprochant de la programmation orientée objet. C'est le cas des langages à type abstraits de données: Simula, ADA, CLU qui partagent tous certaines des propriétés des langages orientés objet. Cependant, ni ADA, ni CLU ne possèdent le mécanisme d'héritage ou celui d'envoi de message. Simula, quant à lui, offre le principe de l'héritage simple [COLN-86].

Diverses extensions de langages usuels ont été adaptés pour permettre un style de programmation par objet. Deux approches ont été suivies.

- Les langages hybrides

  Ces langages ont été créés à partir des langages conventionnels auxquels

les concepts orientés objet ont été ajoutés. Ces langages supportent le mécanisme d'héritage simple. Il s'agit principalement de C++ et Objective-C qui sont des extensions du langage C (appelés aussi pré-processeurs objets de C), de Object Pascal et Clascal, extensions du langage Pascal.

La raison de leur création a été de pouvoir utiliser un langage existant en introduisant un style de programmation fondé sur les objets. En effet, ils ont l'avantage de pouvoir être utilisés quand un langage orienté objet pur (comme Smalltalk) est inacceptable pour des raisons essentiellemnt de performance, c'est-à-dire quand les techniques traditionnelles de programmation sont nécessaires [BCOX-84], [KRAC-85].

- L'extension de LISP

Ce langage étant facilement extensible, il a été aisé d'y ajouter une couche objet. Ce procédé permet de bénéficier des facilités offertes par les mécanismes d'instanciation, d'héritage et de transmission des messages, tout en conservant la puissance de LISP. Nous trouvons des langages tels que LOOPS, CEYX, ObjvLISP, Formes, Flavors ... [COLN-86], [KRAK-85].

Il existe encore de nombreux autres langages développés dans l'optique objet. Les langages de représentation de données, par exemple, dont la création a été motivée par l'intérêt actuel pour l'intelligence artificielle car la notion d'objet apparaît comme un moyen puissant pour résoudre les problèmes de représentation des connaissances (KRL, LOOPS, Mering, KOOL ...) [COLN-86]. Certains langages ont été influencés par Prolog (LOOKS, ESP ...) [COLN-86], d'autres ont été inspirés de Smalltalk (LOOPS, Mering, ACT1, ACT2) [BRIO-??] qui est le langage objet par excellence et auquel nous consacrons le prochain paragraphe.

3.4 Smalltalk-80

3.4.1 Introduction

Smalltalk est considéré comme le principal langage orienté objet et est par ailleurs le premier à intégrer totalement l'ensemble des concepts et principes propres à la programmation orientée objet décrite dans les points précédents. Il fut créé par Alan Kay dans les années 70 au Xerox Palo Alto Research Center. Smalltalk-80 est considéré comme un langage à part entière mais aussi et surtout comme un environnement interactif de programmation très sophistiqué devenu un modèle du genre, on parlera d'ailleurs du système Smalltalk. Il comprend notamment:

- un environnement interactif multi-fenêtre développé autour d'un écran haute résolution (type bitmap) et d'une souris
- un compilateur du langage Smalltalk-80
- un éditeur de texte
- un ensemble d'outils graphiques
- un débogueur symbolique
- un système de gestion de fichiers
- un système de gestion de processus.

Ce système a largement contribué au développement d'interfaces utilisateur perfectionnées telles que les gestionnaires d'écran et de fenêtres, l'iconisation, etc..., et a popularisé l'usage des souris et des écrans haute résolution.

Il tira parti de propriétés présentes dans d'autres langages: SIMULA, pour les notions de classe et d'héritage, LISP, pour l'interactivité et l'extensibilité, PLANNER et PLASMA pour les notions de message. L'évolution du langage fut marquée par trois versions - Smalltalk-72, Smalltalk-76, Smalltalk-80 - et des versions commerciales sont disponibles sur des stations de travail.

3.4.2 Caractéristiques de Smalltalk-80

"Smalltalk est basé sur un très petit nombre de concepts, définis dans une terminologie inhabituelle à des utilisateurs de langages plus conventionnels" [GOLD-83a] mais qui le rendent cependant assez facile à comprendre. Ceci est sans doute dû à l'uniformité avec laquelle l'orientation objet est réalisée dans le sytème. L'utilisateur peut très vite apprendre à connaître ces concepts et explorer les différentes applications de ceux-ci dans l'environnement Smalltalk. Ce dernier peut être présenté en définissant les cinq termes suivants qui constituent pratiquement l'ensemble du vocabulaire Smalltalk: **objet**, **message**, **classe**, **instance** et **méthode**.

Quatre principes, ou axiomes, suffisent à décrire le langage [COLN-86]:

3.4.2.1 Premier principe: toute entité Smalltalk est un objet

Plus que tout autre langage, Smalltalk adhère à ce principe. Littéralement tout est un **objet**, non seulement les entités de haut niveau telles que les fenêtres, directoires, etc..., mais aussi les entités de très bas niveau telles que les entiers, réels, etc... . En réalité, l'objet constitue le seul mécanisme de structuration de données.

Un objet a la capacité de stocker de l'information, de la manipuler et d'exécuter certaines opérations. Il est constitué d'un espace mémoire privé (partie statique de l'objet) et d'un ensemble d'opérations (partie dynamique de l'objet), comme défini au point 3.2.2.1.

L'ensemble des opérations associées à un objet est appelé l'interface de l'objet. Une propriété importante d'un objet est que sa mémoire privée n'est "visible" et ne peut être manipulée que via les opérations appartenant à son interface. Ainsi, par exemple, nous pourrions envisager un objet *farde* dont la mémoire privée serait composée de son *nom* et de son *contenu* . Cet état ne pourrait être manipulé que par l'intermédiaire des opérations de son interface telles que: *afficherNom*, *afficherContenu*, *changerNom(unNom)*, *nom.* On pourrait représenter cela de la manière présentée à la figure 3.4.

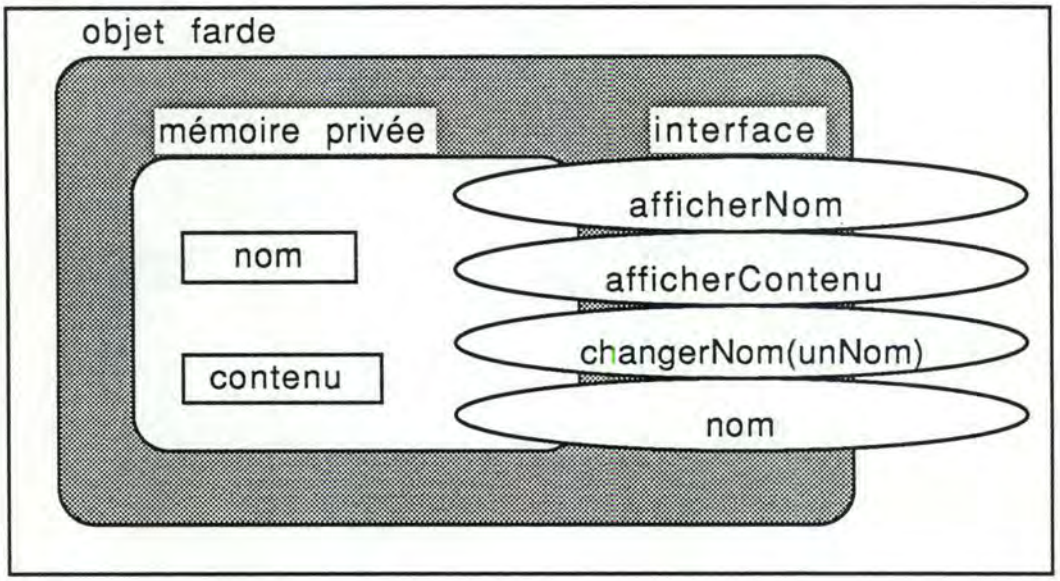


figure 3.4: Schématisation d'un objet Smalltalk

3.4.2.2 Deuxième principe: tout objet est activé à la réception d'un message

De même que Smalltalk ne possède qu'une seule structure de données, il ne possède qu'une seule structure de contrôle, la transmission de **messages** entre objets (point 3.2.2.3).

Ainsi, si l'on reprend l'exemple de l'objet farde (point 3.4.2.1) et que l'on désire connaître le contenu de l'objet farde, il suffira de lui envoyer le message afficherContenu. Ce message ne détermine pas du tout comment accomplir cet affichage, c'est le récepteur, l'objet farde en l'occurrence, qui en a la responsabilité.

Du fait qu'ils spécifient les opérations demandées et non pas comment elles doivent s'exécuter, les messages assurent la modularité du système. On pourrait, en effet, envisager que des fardes, tiroirs, armoires... aient des structures de représentation internes tout à fait différentes tout en répondant au même message afficherContenu. Cette opération devra être impléméntée de manière différente suivant le type de l'objet. Cependant ces aspects internes des objets seront cachés à l'utilisateur.

D'un point de vue plus syntaxique, "les messages sont décrits par des expressions qui sont elles-même des séquences de caractères devant être conformes à la syntaxe du langage de programmation Smalltalk" [XRLG-81]. Une expression est de la forme suivante [KRAK-85]:

<objet> <message>

où le message comprend un sélecteur et éventuellement des arguments. Le sélecteur sert à l'objet receveur pour identifier l'opération qu'il doit exécuter à l'aide des arguments. Un message peut prendre une des deux formes ci-dessous:

<sélecteur>

ou

<sélecteur-1>: <argument-1> ... <sélecteur-n>: <argument-n>

Dans la seconde forme, le sélecteur du message est la concaténation des mots-clés <sélecteur-1>:, <sélecteur-2>:, ... Chacun des mots-clés permet d'identifier l'arguments qu'il précède.

Selon la forme utilisée, on a des messages de types différents. On distingue des messages unaires, des messages binaires et des messages mots-clés:

- un message *unaire* (unary message) correspond à la première forme et est constitué d'un seul sélecteur sans argument. Un exemple est l'expression

farde afficherContenu

où le message afficherContenu est envoyé à l'objet farde.

- un message *binaire* (binary message) a un seul argument et un sélecteur qui est un des sélecteurs binaires spéciaux composés d'un ou deux caractères. Par exemple, les symbôles arithmétiques classiques (+, -, *, /) sont des sélecteurs binaires. On pourrait alors écrire

origine + distance

qui consiste à demander à l'objet origine d'additionner la valeur de l'objet distance à sa valeur.

- un message *mots-clés* (keywords message), quant à lui, possède un ou plusieurs arguments et un sélecteur composé d'une série de mots-clés, un par argument. Un mot-clé est un identifiant suivi du caractère ' : '. On pourrait, par exemple, avoir le message changerNom: possédant un seul argument 'fardeImpôts' qui est le nouveau nom à affecter au récepteur. On écrirait ceci:

farde changerNom: 'fardeImpôts'

Dans le cas bien précis de cet exemple, l'argument est ce qu'on appelle, dans la terminologie Smalltalk, un string, c'est à dire une suite de caractères entourés de simples quotes (').

Un autre exemple pourrait être l'expression suivante:

liste at: index put: anObject.

qui consiste à envoyer le message at: index put: anObject à l'objet liste. Ce message possède deux mots-clés: at: et put: et deux arguments: index et anObject.

Le sélecteur de ces messages est formé par la concaténation de l'ensemble des mots-clés. Ainsi, par exemple, le sélecteur du message at: index put: anObject est at:put: .

Il est à remarquer également qu'un argument peut être une expression. En effet, si l'on a, à sa disposition, deux objets farde1 et farde2, et que l'on veut affecter le nom de farde2 à farde1, on peut écrire:

farde1 changerNom: (farde2 nom)

3.4.2.3 Troisième principe: tout objet Smalltalk est une instance d'une classe

"Une **classe** décrit l'implémentation d'un ensemble d'objets qui représentent tous le même type de composant du système" [GOLD-83a], c'est-à-dire un ensemble d'objets ayant un comportement commun. D'après le premier principe, on dira qu'une classe est aussi un objet. Un classe possède un nom unique qui commence toujours par une lettre majuscule et qui permet de l'identifier parmi toutes les classes du sytème. Les objets individuels décrits par une classe sont appellés ses **instances** et sont créés par un mécanisme d'instanciation. Cela est réalisé par l'envoi du message **new** à la classe désirée. De cette manière, l'expression

farde <- Farde new

où le symbole <- représente l'assignation classique, signifie que l'on crée une nouvelle instance de la classe Farde et que l'on affecte cette instance à l'objet farde. La plupart des classes du sytème répondent à ce message new.

En Smalltalk-80, chaque objet est une instance d'une classe, même un objet représentant un composant unique du sytème est implémenté comme étant une instance d'une classe. Une classe décrit la forme de la mémoire privée de ses instances, ainsi que la manière dont elles exécutent leurs opérations.

Les instances d'une classe sont similaires en ce qui concerne leurs propriétés publiques et privées. Les propriétés publiques d'un objet sont les opérations qui forment son interface. Toutes les instances d'une classe ont le même interface étant donné qu'elles font toutes partie de la même classe. Quant aux propriétés privées, il s'agit d'un ensemble de variables qui constituent sa mémoire privée. Ces variables sont appellées variables d'instance et sont utilisées pour distinguer une instance des autres instances d'une même classe. Parmi ces propriétés, on trouve également des méthodes qui décrivent ce qui doit se passer lorsqu'une instance reçoit un message. A chaque type de message susceptible d'être reçu par l'objet correspond une méthode. Les instances d'une classe utilisent toutes le même ensemble de méthodes pour décrire leurs opérations. Ces variables d'instance et méthodes ne sont accessibles que par l'intermédiaire de l'interface.

On peut donner le format général de la définition d'une classe en Smalltalk (figure 3.5). La notion de super-classe sera expliquée dès l'exposé du quatrième principe. Remarquons encore que l'on fait une distinction entre les méthodes de classe et les méthodes d'instance. Les premières correspondent à des messages envoyés à la classe elle-même et servent généralement à la création et à l'initialisation des instances. Les secondes permettent de répondre à des messages envoyés aux instances de la classe. Quant aux variables de classe, il s'agit de variables dont les valeurs sont partagées par toutes les instances de la classe.

Le format général de la définition d'une classe se présente donc comme présenté à la figure 3.5; les mots en italique seront remplacés par les identifiants et méthodes appropriées à la classe définie.

nom de classe	*un identifiant*
superclasse	*un identifiant*
variables d'instance	*un identifiant un identifiant un identifiant*
variables de classe	*un identifiant un identifiant un identifiant*

méthodes de classe

méthode

méthode

méthode

méthodes d'instance

méthode

méthode

méthode

figure 3.5: Gabarit général de la définition d'une classe

A titre d'exemple, voici un exemple de description de classe en Smalltalk (figure 3.6).

```
class name                      ArticleDuStock

superclass                      Object

instance variable names         reference
                                désignation
                                prixUnitaire
                                quantiteDisponible

class variable names            SeuilDeRupture

instance methods

    commande: quantite par: unClient

        quantite <= (quantiteDisponible - SeuilDeRupture)
          ifTrue: [ self enlever: quantite.
                    self facturer: (self prixTc) * quantite
                         a:    unClient    ]
          ifFalse: [ self ruptureDeStock ]
```

figure 3.6: Un exemple de description de classe

3.4.2.4 Quatrième principe: toute classe est sous-classe d'une autre classe

Le principe d'héritage spécifique de la programmation orientée objet évoqué au point 3.2.2.2 est également présent dans le langage Smalltalk-80. Cependant, étant donné sa particularité, nous allons lui consacrer entièrement le point suivant.

3.4.3 L'héritage en Smalltalk-80

3.4.3.1 Principes

Un sous-ensemble d'objets appartenant à une même classe et ayant un comportement particulier peut être décrit par une nouvelle classe, déclarée **sous-classe** de la première. Celle-ci est alors appelée la **super-classe**.

"Une sous-classe spécifie que ses instances seront les mêmes que sa super-classe, excepté pour les différences explicitement spécifiées" [GOLD-83a].

Une sous-classe est une classe à part entière et peut avoir aussi un ensemble de sous-classes. Chaque classe a une super-classe, bien que plusieurs classes peuvent partager la même super-classe. De cette manière, les classes d'un système

forment un arbre dont la racine est la classe Object, la seule à ne pas posséder de super-classe. Cette classe Object décrit les caractéristiques minimales de tous les objets; ainsi toute classe sera au moins une sous-classe d'Object.

La figure 3.7 présente un fragment de l'arbre d'héritage des classes de base du système Smalltalk.

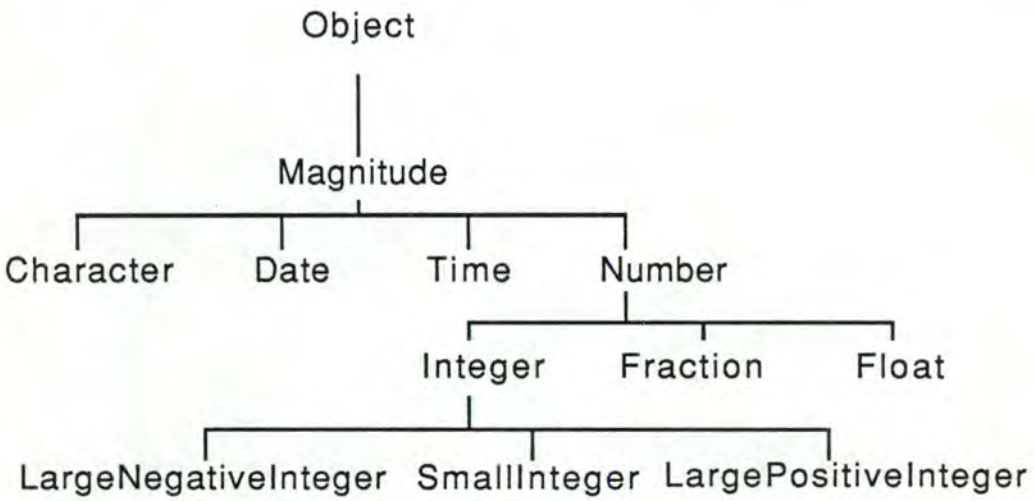


figure 3.7: Partie de l'arbre d'héritage Smalltalk

Une classe fait partie d'une chaîne de classes desquelles elle hérite à la fois les variables et les méthodes. Cette chaîne commence avec sa super-classe et se poursuit avec la super-classe de sa super-classe, et ainsi de suite. La chaîne d'héritage continue à travers les relations de super-classe jusqu'à ce que la classe Object soit rencontrée. Dans l'exemple ci-dessus, la classe Integer hérite ainsi des propriétés des classes Number, Magnitude et Object.

La construction de classes par création de sous-classes peut se faire selon deux axes qui ne sont pas incompatibles:

- ajouter des variables d'instance dans la sous-classe. Ces variables doivent avoir un nom différent de ceux des variables d'instance des superclasses;

- ajouter des méthodes dans la sous-classe. Certaines de ces méthodes peuvent redéfinir des opérations d'une super-classe. A un même nom d'opération peut donc correspondre plusieurs méthodes, c'est à dire plusieurs implémentations (certaines différentes) de cette opération. Ce mécanisme s'appelle la surcharge.

3.4.3.2 Détermination de la méthode à appliquer

De manière générale, la recherche de la méthode à activer pour répondre à un message envoyé à un objet commence toujours dans la classe de l'objet. Si le sélecteur correspondant ne s'y trouve pas, la recherche se poursuit alors dans la super-classe de la classe. Cette recherche suit la chaîne des super-classes et se termine dans la classe Object. Si aucune méthode n'est trouvée, le message

doesNotUnderstand: est envoyé à l'objet récepteur du message avec en argument le sélecteur de la méthode ayant provoqué l'échec.

Ce mécanisme général peut être quelque peu modifié par l'utilisation dans les méthodes, des deux pseudo-variables self et super dont nous n'exposerons cependant pas le fonctionnement sous peine de sortir de notre propos.

3.4.3.3 Les métaclasses

Selon le premier principe, tout objet est instance d'une classe, donc "les classes sont instances de classes particulières appelées métaclasses" [COLN-86]. Les métaclasses sont aux classes ce que les classes sont aux instances. En tant qu'objets, les classes peuvent recevoir des messages, et y répondre grâce aux méthodes définies dans les métaclasses. Ces méthodes servent entre autres à la création des instances et à leur initialisation, ainsi qu'à la modification dynamique du dictionnaire d'une classe.

A chaque classe C est associée automatiquement et implicitement, dès sa définition, une métaclasse C-class, ce qui permet de définir un comportement particulier à la classe C qui est son unique instance. Toutes les métaclasses sont elles-même instances d'une même classe, appelée METACLASS qui spécifient leur comportement commun. Pour assurer la cohérence du modèle, METACLASS est définie comme une instance de METACLASS-class, elle-même instance de METACLASS. Le schéma résultant est représenté à la figure 3.8 .

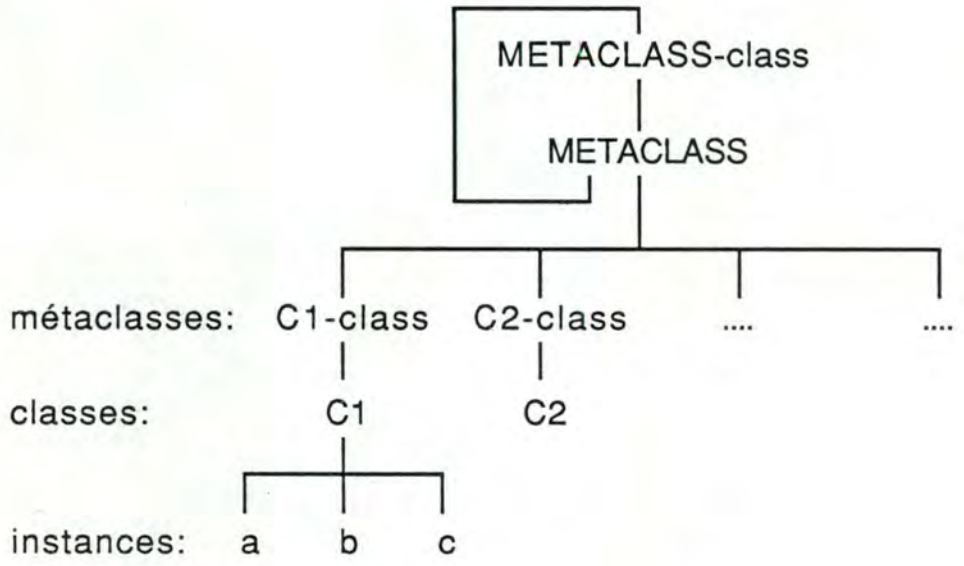


figure 3.8: L'arbre d'instanciation en Smalltalk-80

3.4.4 Les méthodes

Une méthode Smalltalk est déclarée dans la classe de l'objet et est commune à toutes les instances de la classe. Elle décrit comment un objet réalise une opération.

Plus précisément, "Une méthode décrit une séquence d'actions à accomplir quand un message contenant un sélecteur particulier est reçu par une instance d'une classe particulière" [XRLG-81]. Ces actions sont représentées par des expressions Smalltalk séparées par des points (.), ces expressions sont évaluées séquentiellemnt et de gauche à droite. Elles peuvent spécifier certains changements à la mémoire privée de l'objet et/ou peuvent envoyer d'autres messages à d'autres objets. Une méthode peut également renvoyer un objet à l'objet récepteur du message qui l'a déclenchée. Cet objet renvoyé est considéré comme étant la valeur de l'expression qui a envoyé le message ayant déclenché la méthode.

Exemple: si on reprend l'exemple de la classe Farde du point 3.2.2.1, on pourrait écrire la méthode devant être activée lors de la réception du message changerNom: unNom de la manière suivante:

```
changerNom: unNom

    | nom <- unNom
```

qui consiste à affecter unNom à la variable d'instance nom de l'objet receveur du message. Quant au signe ' | ', il provoque la terminaison de la méthode et indique que la valeur renvoyée par la méthode est la valeur de la variable d'instance nom.

Les méthodes peuvent également accéder aux variables d'instance propres à l'objet mais pas à celles d'un autre objet, si ce n'est par l'interface de celui-ci.

Exemple: si un objet envoit un message à l'instance farde1 de la classe Farde de l'exemple précédent, la méthode déclenchée pourra accéder directement aux variables d'instance de farde1 mais non à celles d'une autre instance, farde2 par exemple. Elle ne pourra le faire que par l'intermédiaire des messages appartenant à l'interface de cette deuxième instance.

En Smalltalk, l'unique structure de contrôle est l'envoi de messages à des objets. Cependant cette structure offre la possibilité de définir des constructions telles que celles rencontrées dans les langages classiques. Ces constructions sont les boucles, les alternatives et les itérations. Grâce à ces constructions, un programmeur peut définir, dans ses méthodes, des séquences complexes d'expressions Smalltalk. Mais avant d'analyser chacune de ces constructions, il est nécessaire d'examiner le concept de bloc.

3.4.4.1 Les blocs

L'élément de base des structures de contrôle en Smalltalk est le concept de **bloc**. "Un bloc est une structure permettant au programmeur de retarder l'évaluation d'une suite d'expressions. Il est composé d'expressions séparées par des points et encadrées par des crochets. Un bloc est aussi un objet Smalltalk et peut donc être affecté à une variable" [COLN-86].

Exemple:
```
initBloc <- [array at: index put: elt.
             index <- index + 1]
```

Dans le cas de l'exemple ci-dessus, le code contenu dans le bloc n'est pas évalué et devient la valeur de la variable initBloc. Pour que l'expression soit acceptée, il faut également que les variables array, index et elt aient été déclarées au préalable. L'évaluation est provoquée par l'envoi du message **value** à ce bloc.

Exemple: initBloc value

Les blocs peuvent être paramétrés et constituer de véritables procédures qu'il est possible de donner en arguments aux messages. Les paramètres sont précisés en début de bloc et sont séparés du corps du bloc par une barre verticale. Leur nom commence par le caractère " **:** ".

Grâce à ces blocs, on peut définir des constructions semblables aux structures de contrôle utilisées dans les autres types de langages. Elles sont implémentées sous forme d'expressions dans lesquelles les receveurs ou arguments des messages sont des blocs. Nous allons en voir trois qui seront indispensables dans la suite de notre travail.

3.4.4.2 La sélection conditionnelle: le message ifTrue: ifFalse:

La sélection conditionnelle est un mécanisme qui permet de sélectionner tel ou tel bloc en fonction de la valeur d'une condition. Ce mécanisme est similaire au if... then... else... des langages de type Pascal.

Sa forme générale est la suivante:

condition ifTrue: bloc1 ifFalse: bloc2

où condition est un des deux booléen true ou false, et où bloc1 et bloc2 sont des blocs.

Dans cette expression, le message ifTrue: ifFalse: est envoyé au booléen condition. La méthode déclenchée par la réception de ce message provoque l'exécution du bloc 1 si le receveur est true et provoque l'exécution du bloc2 si le receveur est false.

Exemple: (a < b) ifTrue: [a <- a + 1]
ifFalse: [b <- b + 1]

dont l'équivalent en Pascal pourrait s'écrire:

```
if a < b then a := a +1
          b := b + 1
```

Dans cet exemple, (a < b) est le receveur du message dont le sélcteur est **ifTrue: ifFalse.** La méthode déclenchée exécute les expressions du premier bloc si a est inférieur à b. Si a est supérieur ou égal à b, elle exécute les expressions du second bloc. L'exécution des expressions d'un bloc sera provoquée par l'envoi du message value à ce bloc.

On peut trouver deux autres formes de sélection conditionnelle qui ne fournissent qu'une seule alternative.

Exemple:

- (a < b) ifTrue: [a := a + 1]

- (a < b) ifFalse: [b := b + 1]

3.4.4.3 La boucle

La boucle évalue les expressions d'un bloc aussi longtemps que la valeur d'une certaine condition est vraie. Elle est similaire aux while et until des langages de type Pascal.

Le premier message a comme sélecteur **whileTrue:**. Le message doit être envoyé à un bloc et prend également un bloc comme argument. la forme générale de la boucle est la suivante:

bloc1 whileTrue: bloc2

où bloc1 et bloc2 sont des blocs.

A chaque itération, la méthode activée par le récepteur évalue les expressions du bloc bloc1 et détermine si on doit continuer sur base du résultat de la dernière expression. Si la valeur du receveur est true, alors on exécute les expressions du bloc bloc2 passé en argument.

Exemple: [i < 10] whileTrue: [sum <- sum + (a at: i).
i <- i + 1]

dont l'équivalent en Pascal pourrait s'écrire:

```
while i < 10 do
  begin
    sum := sum + a[i]
  end
```

Le message **whileFalse**: existe également mais exécute le bloc en argument aussi longtemps que la valeur du receveur est false.

3.4.4.4 L'itération sur une collection d'objets

Le système Smalltalk possède un ensemble de classes de base dont la classe Collection qui spécifie les messages appropriés pour manipuler des collections d'objets. Elle possède de nombreuses sous-classes qui permettent de représenter des types de collections spécifiques: Bag, Set, OrderedCollection, ...

La grande majorité de ces classes possèdent une méthode donnant à leurs instances la possibilité de répondre au message **do:** . L'argument de ce message est

un bloc. Lorsqu'une instance de la classe Collection ou d'une de ses sous-classes reçoit ce message, elle évalue le bloc pour chacun des éléments de la collection.

Exemple:

soit listeInfo, une collection contenant des objets informationnels. Supposons que l'on veuille classer ces objets dans l'objet dossier grâce au message classerDans: . Ceci peut se faire de la manière suivante:

```
listeInfo do: [:info | info classeDans: dossier]
```

Dans une énumération, tous les blocs passés en argument sont paramétrés. A chaque itération sur la collection, le paramètre reçoit la valeur de l'élément courant de la collection.

3.4.5 L'interface de programmation

Comme nous l'avons indiqué au point 3.4.1, Smalltalk-80 est nettement plus qu'un simple langage. Il s'agit également d'un environnement de programmation très développé et a notamment contribué au développement d'interfaces utilisateur perfectionnées. C'est justement l'interface du système utilisé que nous allons étudier dans ce point. Notre propos n'est pas du tout de passer en revue les moindres détails de cet environnement mais plutôt d'en exposer les grandes lignes et les principes d'utilisation, de la sorte on pourra percevoir la philosophie mais également l'emploi de notre logiciel puisque celui-ci a été développé dans l'environnement Smalltalk et en reprend donc les mécanismes de base. Les lecteurs interessés par de plus amples explications pourront consulter [GOLD-83b], [TESL-81], [DIED-87] et [GOLD-?].

Un utilisateur et l'environnement de programmation Smalltalk-80 interagissent grâce à un écran du type bitmap, un clavier et une souris. L'écran est utilisé pour présenter de l'information graphique et textuelle à l'utilisateur. Le clavier est utilisé pour introduire de l'information textuelle dans le système. La souris, quant à elle, est utilisée pour sélectionner de l'information sur l'écran, cette information peut être du texte ou des options dans des menus. L'emploi de cette souris est similaire à celui qui en est fait dans tous les systèmes utilisant ce mécanisme.

L'écran Smalltalk contient des rectangles appelés *vues* ou *fenêtres.* Ces fenêtres peuvent être très nombreuses; elles sont créées, déplacées, agrandies, rétrécies et fermées grâce à la souris. Elles peuvent être multiples et se chevaucher, et on peut passer d'une fenêtre à une autre pour exécuter des tâches qui peuvent être liées ou tout à fait distinctes. Les fenêtres peuvent être de plusieurs types toutefois seules deux d'entre elles sont essentiellement interessantes en ce qui nous concerne. Elles portent le nom de *workspace* et de *system browser* (figure 3.9).

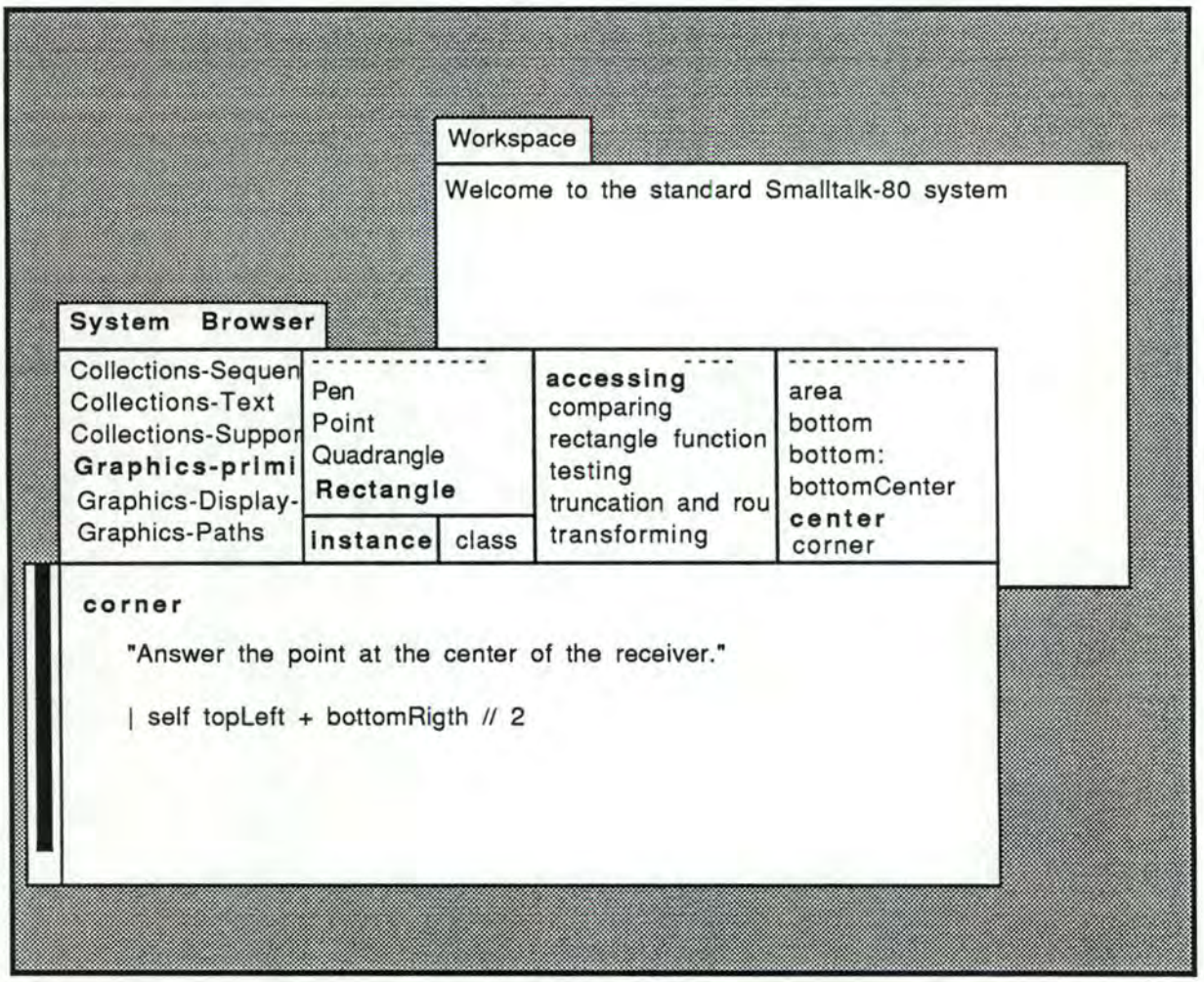


figure 3.9: L'écran Smalltalk: vue générale

a) Les fenêtres System Browser

Rappelons que la programmation orientée objet et particulièrement la programmation en Smalltalk-80 consiste surtout à ajouter de nouvelles classes d'objets et les messages associés, à créer des objets qui communiquent entre eux par envois de messages. Dès lors, être capable de se déplacer facilement à travers la hiérarchie des classes du système et travailler sur des classes est absolument nécessaire pour rendre aisée la programmation en Smalltalk-80.

Les classes du système sont organisées en hiérarchie avec la classe Object comme racine (point 3.4.3). Cependant, on ne doit pas se rappeler continuellement la hiérarchie exacte tout en programmant car grâce à la fenêtre System Browser, on dispose d'un outil très utile pour parcourir cette hiérarchie, consulter chacune des classes et leurs messages ainsi qu'éventuellement les modifier (figure 3.10). Remarquons toutefois que ces classes et messages peuvent également être modifiées par l'intermédiaire d'une multitude d'autres sortes de fenêtres que nous ne verrons pas ici.

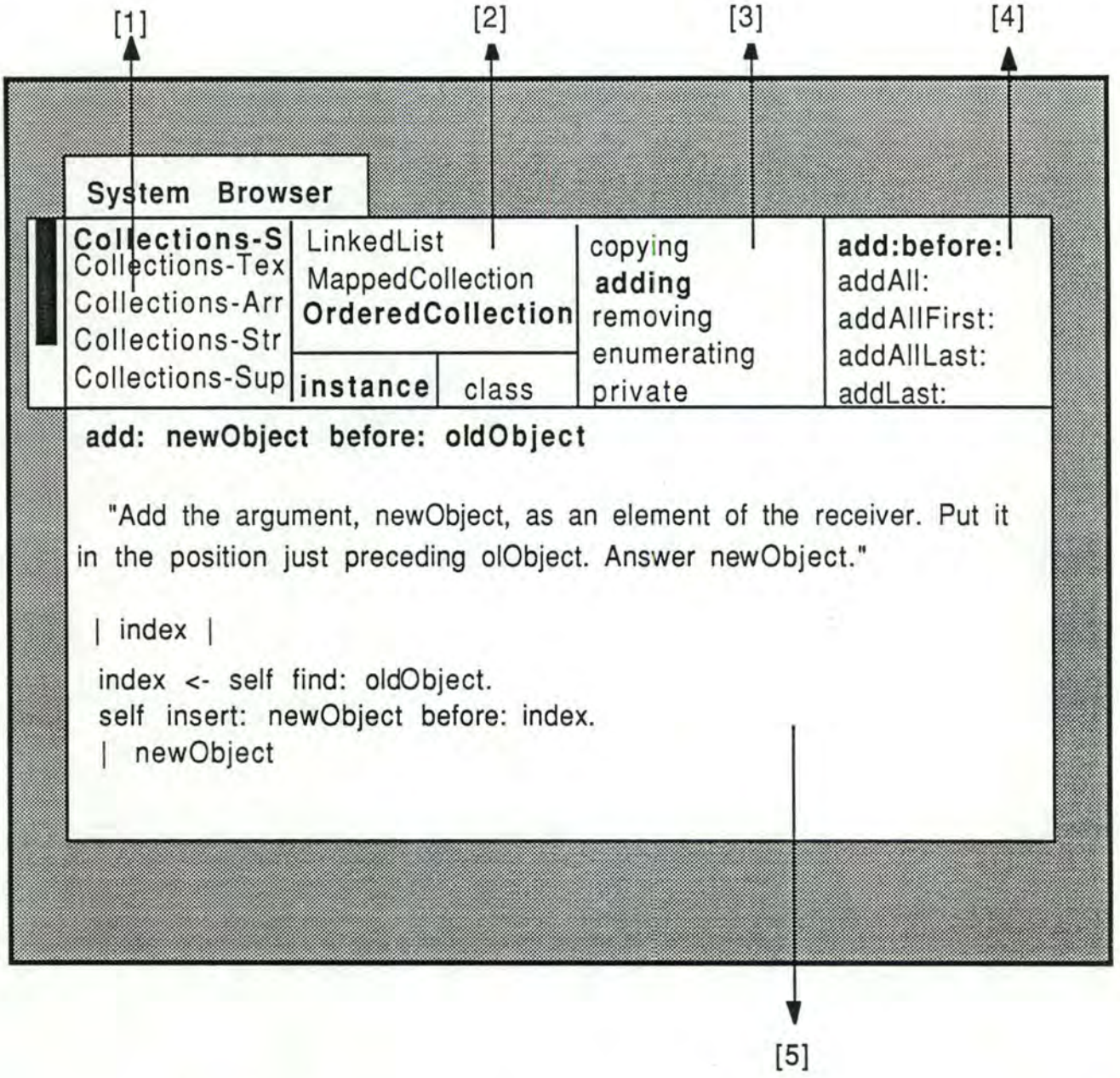


figure 3.10: La fenêtre System Browser

La fenêtre System Browser est composée de cinq fenêtres, chacune d'entre elles disposant d'un pop-up menu particulier accessible grâce aux boutons de la souris:

- la deuxième [2] contient le nom des classes du système
- la quatrième [4] contient les sélecteurs des messages
- la première [1] et la troisième [3] contiennent des catégories de classes et de messages de types semblables
- la cinquième [5] est en quelque sorte un éditeur de texte qui permet d'afficher entre autres la définition de la classe ou de la méthode sélectionnée, de la modifier ou d'en éditer de nouvelles.

Dans la figure 3.10, la catégorie de classes sélectionnée est

Collections-Sequenceable (Collections-S dans le dessin). Une classe particulière, OrderedCollection est sélectionnée dans la deuxième fenêtre.

Quant aux deux cases instance/class au bas de la seconde fenêtre, elles fonctionnent comme ceci: si la case instance est sélectionnée - comme dans la figure 3.10 -, ce qui apparaîtra dans les fenêtres à droite et en dessous correspondra à des messages pouvant être envoyés à des instances de la classe sélectionnée. Si, par contre, class est sélectionnée, les données affichées correspondront à des messages destinés à être envoyés à la classe elle-même qui est considérée également comme un objet.

Les catégories de messages pouvant être envoyés à des instances de OrderedCollection sont affichés dans la troisième fenêtre. Dans la quatrième, on trouve les sélecteurs de messages. Dans cette fenêtre, on a sélectionné add: before: et la méthode permettant de répondre à un message de cette forme est affichée dans la fenêtre du bas.

Les options présentées dans les pop-up menus des différentes fenêtres permettent de nombreuses possibilités pour naviguer dans l'ensemble de la hiérarchie du système.

b) Les fenêtres Workspace

Les fenêtres de ce type contiennent du texte qui peut être édité ou évalué, c'est-à-dire que l'on peut y taper du texte et l'exécuter comme étant des expressions Smalltalk.

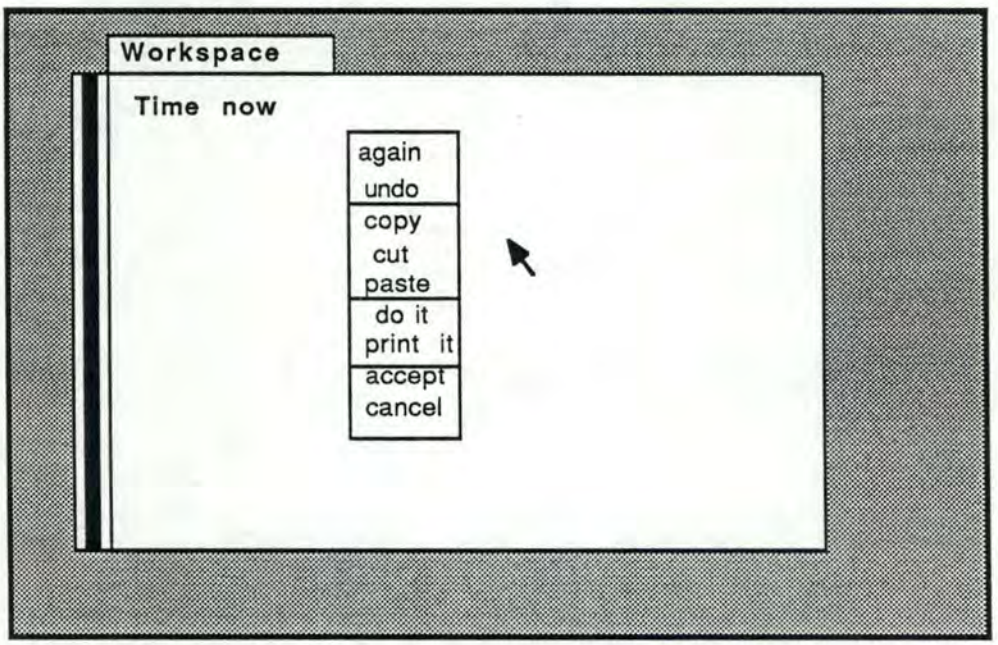


figure 3.11: La fenêtre Workspace

3.5 La méthode de développement par objet

Nous allons maintenant analyser dans les grandes lignes, les principes inhérents au développement de programmes de manière orientée objet et voir en quoi ces principes diffèrent de ceux utilisés par d'autres méthodes plus classiques.

Grâce à leur puissance déclarative et à leur modularité, les langages orientés objet simplifient en grande partie le travail de programmation. Toutefois, pour tirer pleinement profit de ces langages, il est nécessaire de modifier sa manière d'aborder et d'analyser un problème [COLN-86].

En effet, la conception orientée objet peut être définie comme "une technique qui, contrairement à la conception classique (fonctionnelle), base la décomposition modulaire d'un logiciel sur les classes d'objets manipulées par le système et pas sur les fonctions que celui-ci exécute" [MEYE-87b]. On cherche dès lors à cerner les entités qui existent et ensuite à spécifier la manière dont ces entités interagissent plutôt que de définir les fonctionnalités du programme désiré comme dans les approches classiques. En réalité, l'approche par objet tente d'éviter la question des fonctionnalités aussi longtemps que possible. Pour ce faire, on crée des classes d'objets et des opérations associées de manière à implémenter le concept de type abstrait (point 3.2.1).

3.5.1 Le principe d'abstraction

Le principe d'abstraction utilise le principe de "l'information cachée" et peut jouer deux rôles.

Son premier rôle consiste à créer une classe d'objets. Ceci permet de créer une abstraction sur les données en cachant l'implémentation proprement dite de ces dernières, celles-ci n'étant alors accessibles que par l'intermédiaire des primitives appartenant à leur interface.

La création de hiérarchies de classes permet d'associer un autre rôle à ce principe d'abstraction. En effet, il donne la possibilité de cacher à certains programmes, des informations qui sont inutiles à leur fonctionnement.

Afin d'illustrer ce principe, reprenons l'exemple classique de la classe Fenêtre [GOLD-83a]. Supposons que l'on écrive un programme dans lequel on doit manipuler des fenêtres. On pourrait envisager de devoir traîter deux types de fenêtres: des fenêtres où on ne peut pas écrire et des fenêtres où on peut écrire. On pourrait alors créer deux classes: Fenêtre et FenêtreEcriture. La première spécifierait la structure de données (exemple: point supérieur gauche et droit et contenu de la fenêtre) et les messages (ouvrir, fermer, diminuer, ...) communs à toutes les fenêtres. La seconde constituerait une spécialisation de la première et contiendrait dans son interface deux messages particuliers: écrire et effacer. On peut alors les organiser de la manière suivante:

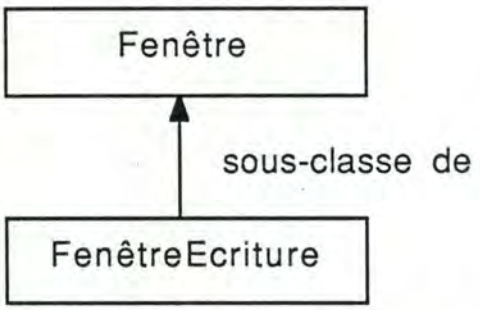


figure 3.12: Hiérarchie de deux classes de fenêtres

où fenêtreEcriture est la sous-classe de Fenêtre et hérite toutes ses propriétés. A partir de ce moment, on pourrait supposer que certains programmes ne s'interessent pas aux fenêtres dans lesquelles on peut écrire. Dans ce cas, ces programmes feront abstraction de la sous-classe FenêtreEcriture. Pour chaque application, on peut ainsi choisir le niveau de précision des informations sur lesquelles on veut travailler.

3.5.2 Le principe de la généralisation

La généralisation peut être considérée comme "une absraction qui permet à un ensemble d'objets individuels d'être pensés et référencés par un concept général unique" [SMIT-77].

Ce principe est fréquemment utilisé pour conceptualiser le monde réel. En effet, à tout moment, à partir de quelques cas isolés, on a souvent tendance à généraliser. Ainsi, si on reprend l'exemple du point 3.5.1, on pourrait être amené à ne considérer que la classe Fenêtre et à ne s'interesser qu'aux informations communes à toutes les fenêtres quel que soit leur type. On fait alors abstraction de tous les détails concernant les fenêtres.

3.5.3 Les avantages de ces deux principes

Lorsque l'on travaille avec une hiérarchie de classes et que l'on applique ces principes d'abstraction et de généralisation, on a tendance à reporter le plus haut possible dans la hiérarchie, des données et/ou des messages communs à l'ensemble des sous-classes. Les autres informations laissées dans les sous-classes, constituent alors des détails propres à ces classes. Celles-ci forment ainsi des spécialisations de leurs super-classes.

De plus, une hiérarchie de données permet d'introduire, à tout moment et de manière contrôlée, des détails dans le système et une grande stabilité pour les applications qui utilisent ces données. En effet, on peut très bien modifier un niveau de la hiérarchie sans que les programmes utilisant les niveaux supérieurs ne s'en rendent compte.

Du fait de la très grande souplesse et modularité des systèmes orientés objets, le domaine des objets existants peut être enrichi à partir d'un noyau de base sans pour autant altérer celui-ci. La modification d'un programme revient alors dans la plupart des cas, à ajouter de nouvelles classes d'objets, héritant des propriétés de classes déjà existantes qui n'ont pas à être modifiées. Dans ces nouvelles classes, certaines

méthodes sont modifiés et d'autres sont éventuellement ajoutées.

Ce genre de modularité vient s'ajouter à celle qu'on trouve habituellement en programmation structurée. Elle permet, en effet, de changer la manière dont un module réalise la tâche qui lui est assignée sans que cela ait une influence sur le reste du programme.

A partir de ces grands principes, on peut déduire une méthodologie de développement qui va tenter de garder la solution aussi proche que possible du problème du monde réel. Dans ce monde, les objets avec lesquels on travaille sont exprimés à l'aide de substantifs et les opérations par des verbes. La méthodologie proposée par [JAMS-84] se base sur le fait que la formulation d'un problème peut être employée pour produire des unités implémentables par identification des objets et des opérations réalisées sur les objets. Elle comporte les étapes suivantes:

1. Définir le problème
2. Développer la stratégie informelle
3. Formaliser la stratégie
   a. Identifier les objets et leurs attributs
   b. Identifier les opérations sur les objets
   c. Etablir les interfaces
   d. Implémenter les opérations

La première étape correspond à la phase d'analyse des méthodes classiques. La deuxième consiste à formuler le problème dans un langage naturel (ex: le français). La formalisation de la stratégie commence par le repérage, dans la stratégie informelle, des objets et des opérations appliquées à ces objets. Ensuite, il s'agit d'établir les interfaces (point 3.2.2.1) et d'implémenter les opérations. Ces deux dernières phases sont malheureusement souvent laissées à l'inspiration du concepteur, la méthodologie ne fournissant pas de règles pour mener à bien ce travail de manière efficace.

3.6 Justification du choix de la programmation orientée objet et de Smalltalk-80

Après ce parcours des méthodes et langages orientés objets, il nous reste à présent à justifier notre choix.

Au chapitre précédent, il a été dit qu'un bureau pouvait être décrit comme un ensemble d'objets ayant des relations entre eux. De plus, des opérations peuvent être spécifiées sur ces objets. Le parallélisme entre cette manière de procéder et celle utilisée par la programmation orientée objet (recensement des entités manipulées et par après des opérations effectuées sur ces entités) devient alors évident et cette dernière semble alors devenir la façon la plus naturelle de représenter un environnement de bureau.

Mais telle n'est pas la seule raison de notre choix. Le principal attrait de la conception orientée objet pour notre travail est sans doute la capacité qu'elle offre de

développer des produits réutilisables et extensibles. En effet, plus encore que de nombreux programmes, le logicielque nous développons sera très souvent soumis aux changements étant donné que d'une étude de bureau à une autre, on sera très certainement obligés de pouvoir manipuler de nouveaux types d'objets (télex, terminal, téléfax,...) et d'effectuer de nouvelles opérations sur ces objets. Ce type de programmation, grâce aux concepts d'objet et d'héritage, s'applique donc tout à fait à notre problème.

Le choix de la méthode de développement se justifie également par le fait que notre objectif est de réaliser un prototype de l'outil proposé. Or, du fait de leur très grande modularité, les langages permettant ce type de programmmation constituent d'excellents outils de prototypage. Un objet peut, en effet, être considéré comme une boîte noire que l'on ne voit uniquement que de l'extérieur. L'adjonction ou le retrait d'un élément du programme n'entraîne alors que des perturbations locales, la connaissance étant distribuée parmi les différents objets.

Quant au langage de programmation, le fait qu'on ait opté pour Smalltalk-80 résulte de deux constatations.

La première est que ce dernier est retenu par de nombreux auteurs comme étant le langage par excellence pour faire du prototypage. Diederich [DIED-87] le définit d'ailleurs de la manière suivante: "Il s'agit d'un outil complet pour encourager au prototypage expérimental". Il permet de réaliser des changements fréquents de l'architecture d'un système, en entraînant généralement peu de reprogrammation et en n'introduisant que des erreurs minimes qui sont facilement identifiées et corrigées.

La raison de ce comportement est bien entendu que Smalltalk-80 reprend tous les principes et concepts propres à la programmation orientée objet en poussant leur utilisation à l'extrême, offrant ainsi une très grande modularité et permettant de faire ce que Diederich [DIED-87] appelle la "programmation intrépide". Ce style de programmation encourage fortement à faire de l'expérimentation par alternatives pour la construction d'algorithmes, de programmes d'application et d'architectures de systèmes sans craindre de s'empêtrer dans une foule de détails inutiles.

En plus d'être rendu possible par l'utilisation du principe d'abstraction de données dans des objets et par le principe d'héritage, le prototypage l'est également grâce à son environnement développé qui permet de travailler facilement avec l'ensemble des classes, méthodes et messages du système.

La seconde constatation est que le Smalltalk-80 propose des classes de base dont l'utilisation permet de créer des interfaces de haut niveau (éditeurs graphiques, multi-fenêtrage, pop-up menu,...), ce qui est primordial quand on veut réaliser un outil dont le principe de base est l'animation graphique.

CHAPITRE 4

L'ARCHITECTURE DU SYSTEME SMALLTALK

CHAPITRE 4: L'ARCHITECTURE DU SYSTEME SMALLTALK-80

Puisque notre logiciel a entièrement été développé dans l'environnement Smalltalk, l'architecture, autour de laquelle il s'articule, a fortement influencé notre travail. Il s'avère donc nécessaire d'étudier attentivement l'architecture de Smalltalk.

Dans le premier point de ce chapitre, nous analyserons la hiérarchie des classes que nous pourrions appeler classes de base et nous nous attarderons quelque peu aux classes qui nous seront utiles dans la suite de notre mémoire. Dans un deuxième point, nous examinerons l'architecture sur laquelle repose l'ensemble du système mais aussi notre logiciel. Dans un dernier point, nous passerons en revue quelques classes plus complexes qui s'insèrent également dans la hiérarchie du système Smalltalk. Ces classes interviendront à la fin de notre travail.

4.1 Les classes de base

4.1.1 La classe Object

Comme nous l'avons dit au point 3.4.3, l'ensemble des classes du système Smalltalk est organisé en un arbre hiérarchique dont la racine est la classe **Object**. Ceci implique que toute classe possède la classe Object parmi ses super-classes. Dans cet arbre, une classe hérite de toutes les variables et messages définis dans chacune de ses super-classes, et donc en particulier des variables et messages définis dans la classe Object.

C'est la raison pour laquelle les messages communs à tous les objets du système sont définis dans la classe Object. De cette manière, n'importe quel objet peut répondre aux messages définis dans cette classe. Ces messages fournissent une réponse pour tous les objets et constituent un point de départ à partir duquel on peut développer de nouvelles classes. Dans ces nouvelles classes, on peut ajouter de nouveaux messages et variables ou modifier des messages hérités des super-classes.

Les messages définis dans la classe Object permettent entre autres:

- d'effectuer certains tests sur un objet (exemple: savoir si un objet est une instance d'une classe donnée, s'il peut répondre à certains messages, ...).
- de réaliser des comparaisons entre objets.
- d'effectuer des copies d'objets.
- d'accéder à des objets.
- d'imprimer des objets.
- de répondre à des messages d'erreurs.

4.1.2 Les sous-classes de la classe Object

Par construction, le système Smalltalk comporte un nombre important de classes. Leur richesse et leur quantité font de Smalltalk un outil très complet. Dans ce point, nous n'avons pas la prétention de les analyser toutes mais simplement de passer en revue les quelques classes qui seront utilisées dans la suite de ce mémoire. Pour un

exposé plus détaillé, voir [GOLD-83a].

Comme nous venons de le voir au point 4.1.1, toutes les classes sont organisées en arbre dont la racine est la classe Object. Une partie de cet arbre apparait à la figure 4.1 .

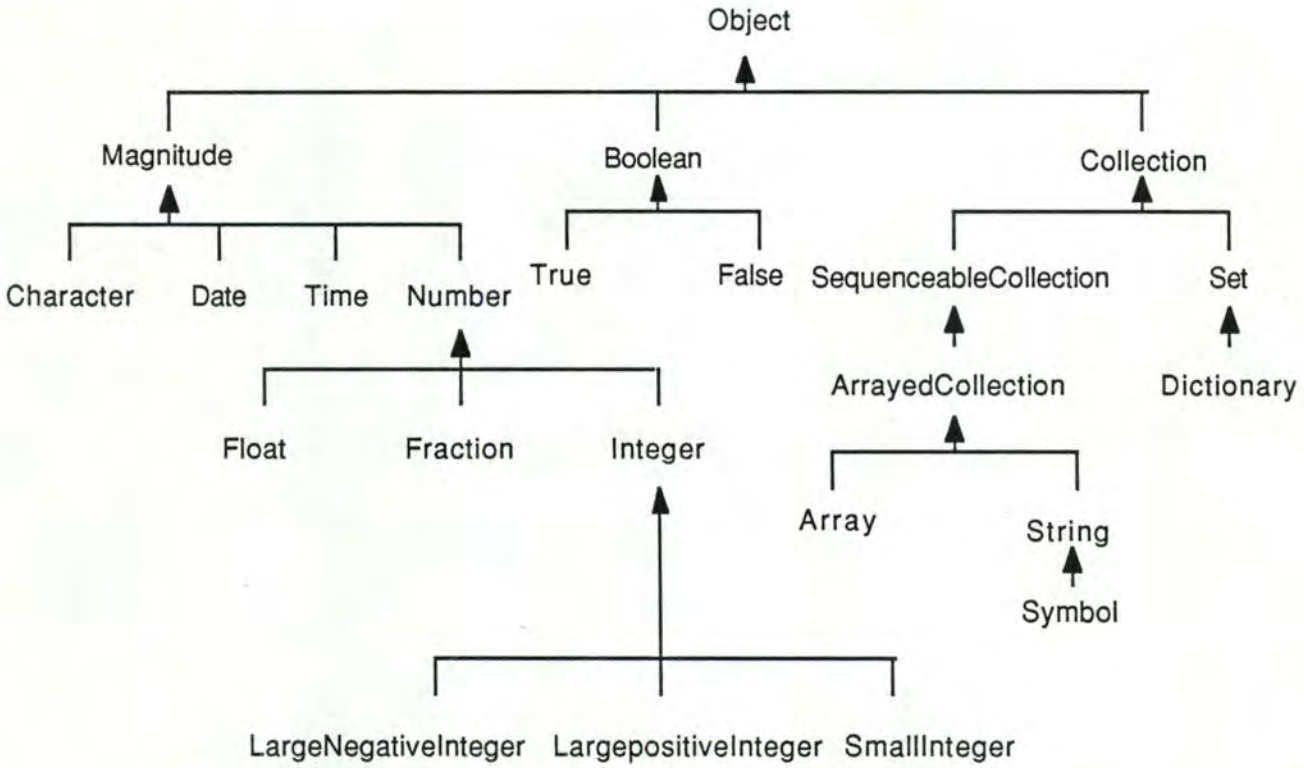


figure 4.1: Une partie de l'arbre d'héritage des classes de base du système Smalltalk-80

Nous nous intéresserons successivement aux classes Number et à ses sous-classes, String, Symbol, Array, Dictionary et Boolean.

4.1.2.1 La classe Number et ses sous-classes

Contrairement au langage Simula dans lequel le principe d'envoi de messages à des objets n'était utilisé que pour des objets de haut niveau, en Smalltalk, il est appliqué à tous les objets et en particulier aux nombres. Toutefois, au fur et à mesure du développement du système, l'envoi de messages à des valeurs numériques a été raffiné et peut maintenant être manipulé de la même manière que dans tout autre langage.

Comme dans tous les systèmes, les valeurs numériques sont utilisées dans la plupart des programmes Smalltalk. Chaque type de nombre est représenté par une classe particulière. Toutes les valeurs numériques doivent pouvoir répondre à des messages communs définis dans la classe **Number**. Ces messages représentent toutes les opérations arithmétiques et de comparaison classiques (+, -, *, /, cos, sin, <, >, ...). Toutefois, la plupart d'entre-eux doivent être redéfinis par les sous-classes de laclasse Number car leur représentation dépend de leur type.

La classe Number possède trois sous-classes: Float, Fraction et Integer. La classe Integer ajoute des messages particuliers aux nombres entiers et possède trois sous-classes: SmallInteger, LargePositiveInteger et LargeNegativeInteger. La première d'entre elles fournit une représentation économique en place mémoire pour des valeurs entières qui apparaissent fréquemment dans des opérations de comptage ou pour les index des tableaux.

Les nombres dans le système Smalltalk sont des instances des classes Float, Fraction, SmallInteger, LargePositiveInteger et LargeNegativeInteger. Les classes Number et Integer ne font que définir des messages partagés par leurs sous-classes mais ne spécifient pas de représentations particulières pour des valeurs numériques. On ne trouvera donc pas d'instances des classes Number et Integer. La hiérarchie des classes définissant les valeurs numériques apparait à la figure 4.2 .

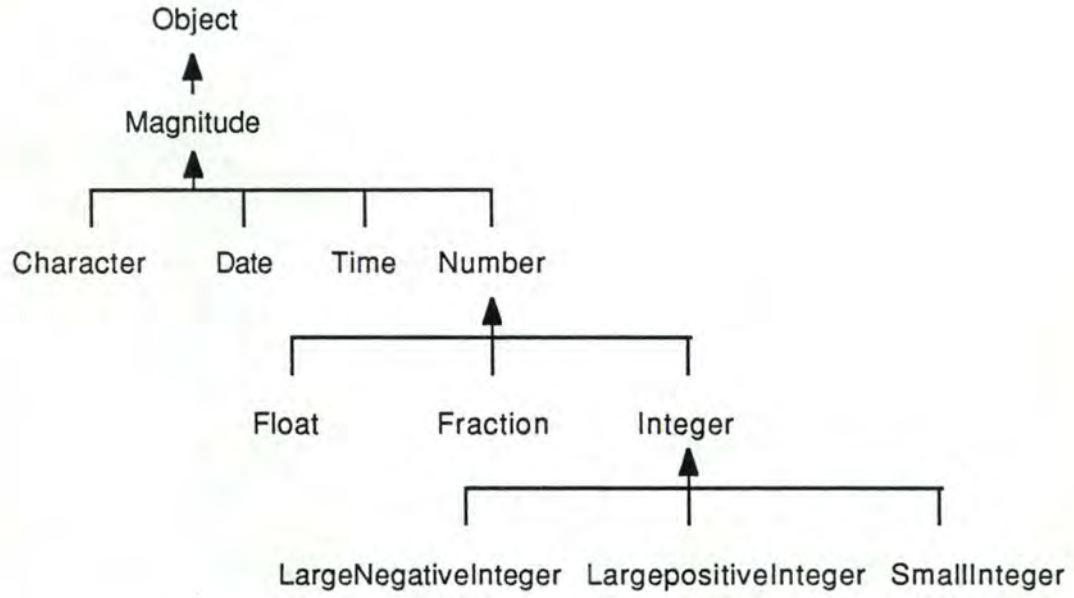


figure 4.2: La classe Number et ses sous-classes

4.1.2.2 La classe Collection et ses sous-classes

En Smalltalk, une collection représente un groupe d'objets. Ces objets sont appelés les *éléments* de la collection. Les collections sont les structures de données de base pour le développement de programmes dans le système Smalltalk-80.

La classe Collection regroupe tous les messages communs à toutes les collections. Elle possède également un grand nombre de sous-classes, chacune d'entre elles définissant un type de collection particulier. La figure 4.3 présente la hiérarchie des sous-classes de la classe Collection.

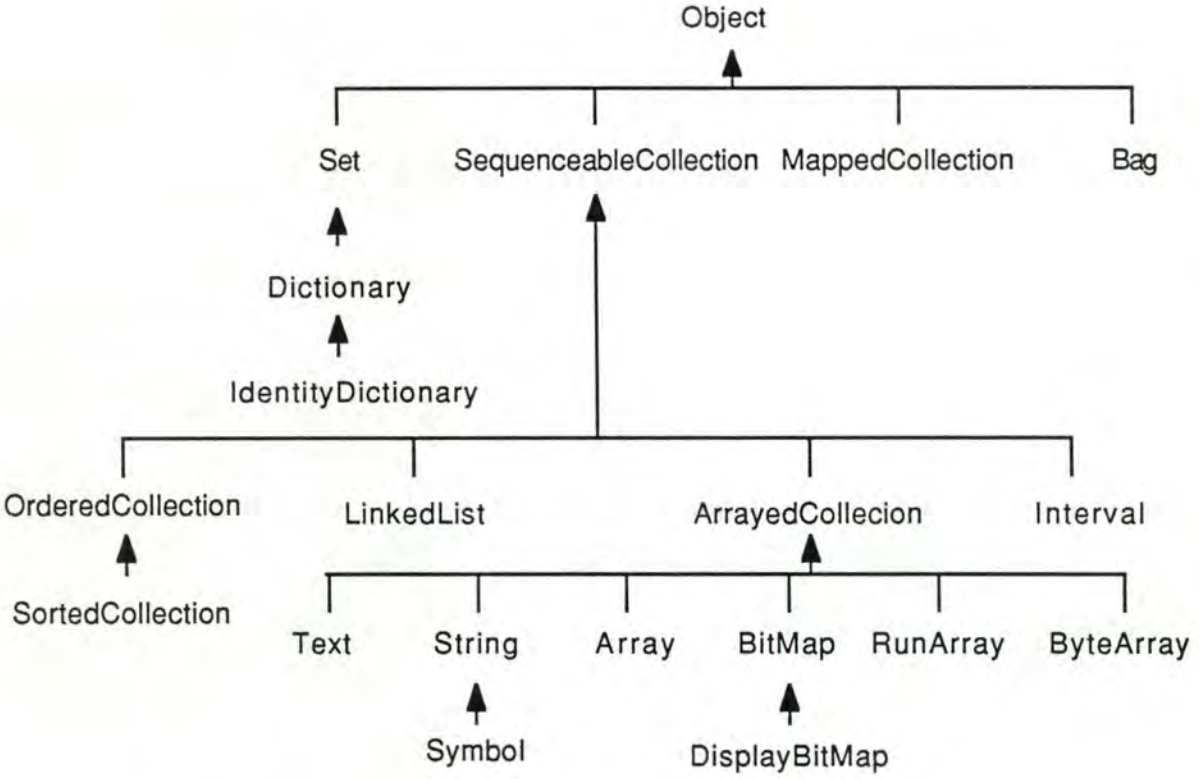


figure 4.3: La classe Collection et ses sous-classes

Tous les messages décrits dans la classe Collection sont compris par n'importe quel type de collection. On trouve quatre catégories de messages:

- des messages pour insérer de nouveaux éléments dans une collection.
- des messages pour supprimer des éléments dans une collection.
- des messages pour tester le présence d'éléments dans une collection.
- des messages pour parcourir tous les éléments d'une collection.

Parmi les nombreuses sous-classes de la classe Collection, nous allons examiner les classes String, Symbol, Array et Dictionary.

a) La classe String

Les super-classes de cette classe sont dans l'ordre: ArrayedCollection, SequenceableCollection et Collection.

Une instance de la classe String est une collection ordonnée dont les éléments sont des caractères. La forme générale d'une telle collection est une suite de caractères entourées de simples quotes ('). A titre d'exemple, 'armoire' représente une instance de la classe String composée des caractères a, r, m, o, i, r, e.

Outre les messages hérités de ses super-classes, les instances de la classe String peuvent répondre à des messages permettant l'accès à des caractères individuels, le remplacement des morceaux de string et la comparaison entre deux strings. N'importe quel caractère peut être inclus dans un string. Si une simple quote (') doit figurer dans un string, il doit être doublé ('') pour éviter la confusion avec les

délimiteurs.

b) La classe Symbol

La super-classe de cette classe est la classe String. Elle hérite donc de toutes les variables de cette classe et de ses super-classes.

Une instance de la classe Symbol est une collection ordonnée de caractères qui est unique dans le système Smalltalk.

La forme générale d'une instance de la classe Symbol est une suite de caractères précédée du sigle #. Par exemple, #armoire est un symbole composé des sept caractères a, r, m,o, i, r, e.

c) La classe Array

Les super-classes de la classe Array sont les mêmes que celles de la classe String. Une instance - que nous appellerons un tableau - de cette classe est une collection ordonnée d'éléments dont le type est quelconque. La forme générale d'une telle collection est une suite de valeurs numériques, de caractères, de strings, de symboles et/ou de tableaux délimitée par des parenthèses et précédée par le sigle de pondération #. Les éléments de la collection sont séparées par des blancs.

A titre d'exemple, l'expression #(9 'armoire' #nomArmoire ('un' 1)) désigne une instance de la classe Array dont les éléments sont dans l'ordre un nombre, un string, un symbole et une autre instance de la classe Array.

Une instance de la classe Array répond à tous les messages définis dans ses super-classes.

d) La classe Dictionary

Les super-classes de la classe Dictionary sont les classes Set et Collection. Une instance de cette classe est une collection d'associations entre des clés et des valeurs. Les éléments d'un dictionnaire sont des instances de la classe Association, une simple structure de données pour stocker et récupérer les membres d'un couple clé-valeur.

Une autre façon de définir un dictionnaire est qu'il s'agit d'une collection dont les éléments ne sont pas ordonnés mais qui ont chacun une clé d'accès spécifique. Dès lors, les éléments d'un dictionnaire sont des objets de type quelconque avec des clés externes.

La classe Dictionary, outre les messages hérités de ces super-classes, définit dans son interface de nombreux messages appropriés aux caractéristiques particulières de ce genre de collection.

A titre d'exemple, supposons que l'on désire créer un dictionnaire appelé contraire dans lequel on trouverait des symbôles et leur contraire. Une manière de procéder serait la suivante:

contraire <- Dictionary new.

"Cette expression crée une instance de la classe Dictionary qu'elle affecte à la variable contraire"

contraire at: #chaud put: #froid.
contraire at: #pousser put: #tirer.
contraire at: #venir put: #aller.
contraire at: #devant put: #derrière.
contraire at: #sommet put: #fond.

"Ces expressions permettent de remplir le dictionnaire contenu dans la variable contraire. Le premier argument des messages est la clé, le second est la valeur."

Le dictionnaire contenu dans la variable contraire se présente maintenant comme suit:

clés	valeurs
#chaud	#froid
#pousser	#tirer
#venir	#aller
#devant	#derrière
#sommet	#froid

Pour accéder à une valeur précise, il est nécessaire d'utiliser la clé correspondante.

exemple: valeur <- contraire at: #devant.

Cette expression affecte à la variable valeur la valeur associée à la clé #devant dans le dictionnaire contraire.

4.1.2.3 La classe Boolean

Les valeurs logiques true et false présentes dans tous les langages de programmation sont des instances des classes True et False. La super-classe de ces deux classes est la classe Boolean. Ses deux sous-classes n'ajoutent aucun nouveau message mais en redéfinissent certains afin d'obtenir de meilleures performances. Dans l'interface de la classe Boolean apparaissent notamment les messages dont les sélecteurs sont ifTrue: IfFalse:, ifTrue: et IfFalse:.

4.2 L'architecture Modèle-Vue-Contrôleur ou M.V.C.

4.2.1 Introduction

Smalltalk-80 est un langage mais aussi, comme nous l'avons vu au point 3.4.5, un environnement de programmation très perfectionné. Cet environnement fait un usage intensif des fenêtres et des menus de tous genres. Il fournit également un ensemble d'outils efficaces pour le développement de programmes à orientation graphique et d'interface-utilisateurs de haut niveau.

De nos jours, des outils de ce genre sont de plus en plus souvent développés autour d'une architecture en trois couches dont les grandes lignes sont exposées dans [BCOX-86]. La figure 4.4 représente cette architecture dont les trois couches sont:

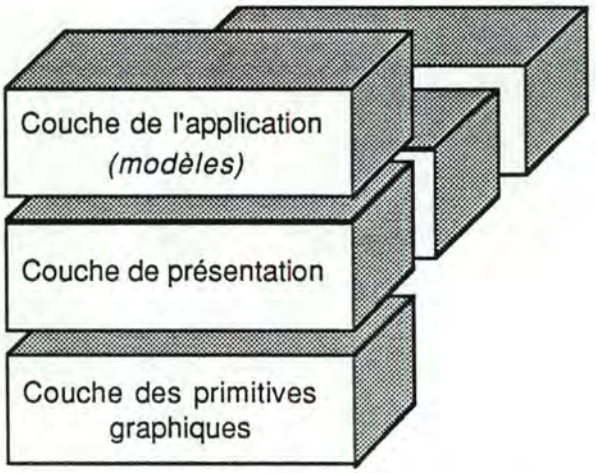


figure 4.4: Les trois couches de l'architecture

1 ***La couche application:*** Cette couche implémente la fonctionnalité du programme à développer, et ne se préoccupe pas de la manière dont les données vont être affichées à l'écran. Les composants de cette couche sont appelés les *modèles* car la fonction de la plupart des programmes est de modéliser certains aspects de la réalité. Ces modèles ne détiennent aucune information concernant l'interface utilisateur. Ainsi, la couche de présentation peut être remplacée à tout moment. Généralement, les modèles sont accessibles à partir des vues mais jamais le contraire.

2 ***La couche de présentation:*** Les composants de cette couche sont des*vues*, chacune d'entre elles implémente un interface-utilisateur pour un modèle spécifique de la couche application. Un modèle donné peut être destiné à différentes catégories d'utilisateurs, dès lors on trouvera généralement différentes vues associées à un seul modèle afin d'assurer une présentation différente des données. Ceci est représenté à la figure 4.4 où plusieurs blocs composent la couche de présentation alors qu'un seul compose la couche application.

3 ***La couche des primitives graphiques:*** Cette couche est constituée d'un ensemble

de primitives graphiques et implémente un interface entre la couche de présentation et le matériel graphique (écran bitmap, ...).

Les vues se chargent donc d'assurer la présentation des modèles aux utilisateurs par l'intermédiaire d'outils tels que des terminaux graphiques et des souris. La vue fait appel au modèle pour obtenir les données, formatte ces données, les affiche à l'écran, reçoit des commandes de l'utilisateur et ordonne au modèle ce qu'il doit faire. Les vues détiennent généralement une référence vers leur modèle mais l'inverse est rarement le cas. Les modèles sont tout à fait passifs et ne détiennent jamais d'information qui pourraient les rendre dépendants de la couche de présentation. Cette dernière devient donc facilement interchangeable.

Dans le système Smalltalk-80, cette architecture introduit un composant supplémentaire appelé *contrôleur* et est connue sous le nom de Modèle-Vue-Contrôleur ou M.V.C.. Sa connaissance est absolument nécessaire pour réaliser un travail interessant en Smalltalk. C'est pourquoi nous allons l'examiner rapidement.

4.2.2 L'architecture Modèle-Vue-Contrôleur

Le système Smalltalk-80 est un environnement et un langage de programmation extensibles (figure 4.5).

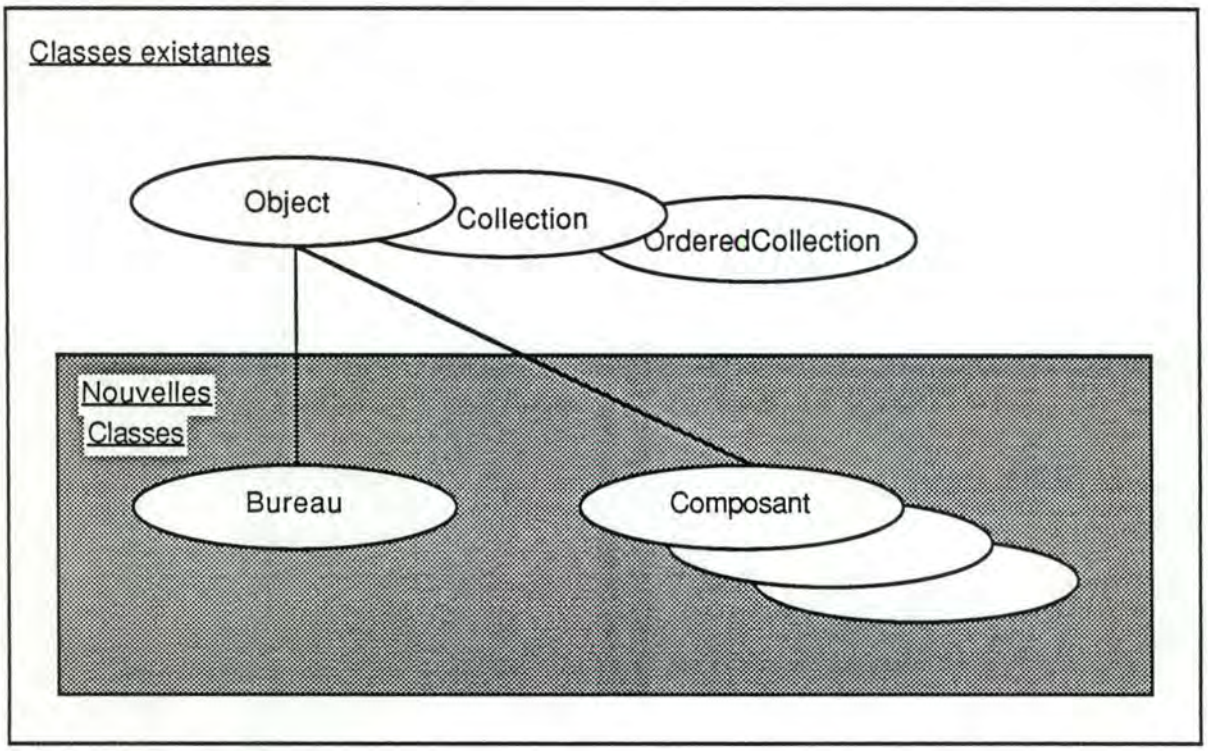


figure 4.5: Extension de l'environnement Smalltalk.

Le développement d'un programme Smalltalk nécessite la création de nouvelles classes d'objets destinées à contenir les méthodes et structures de données du programme à réaliser.Ces nouvelles classes, définies comme des extensions de

classes existantes, permettent d'étendre les capacités de ces classes à un domaine d'application spécifique. L'environnement d'origine reste évidemment disponible pour la construction d'autres programmes.

4.2.2.1 Les composants d'un programme Smalltalk

Lorsqu'il fait un usage sophistiqué des vues, le programme Smalltalk est composé de plusieurs parties, chaque partie étant d'un des trois types suivants: modèle, vue ou Contôleur.

Un *modèle* représente les données du programme et les opérations qui peuvent être exécutées sur ces données.

Une*vue* est le mécanisme utilisé par un modèle pour afficher ses données dans des fenêtres, à l'écran. Elle sert d'intermédiaire entre le modèle, source d'information, et l'écran.

Un *contrôleur* a la responsabilité d'interpréter les commandes de l'utilisateur en fonction de la vue qu'il contrôle et de déclencher une réponse correspondante de la part de la vue ou du modèle selon les cas.

Le modèle représente donc les données du programme et les messages pour y accéder, tandis que la vue et le contrôleur représentent l'interface-utilisateur.

4.2.2.2 La construction d'un programme

La construction d'un programme Smalltalk peut se faire en trois étapes correspondant chacune au développement d'une des composants du programme.

Première étape: la conception et le test du modèle du programme

Un modèle représente les données d'un programme et les opérations qui peuvent être exécutées sur ces données. Un modèle doit coopérer avec les vues et les contrôleurs en obéissant à des protocoles (point 3.2.2.3) déterminés en vue de fournir les données destinées à être affichées à l'écran. Toutefois, le modèle doit être implémenté de la manière la plus simple possible. Il ne doit détenir aucune information concernant l'interface-utilisateur de manière à rester le plus indépendant possible de ses vues et contrôleurs. Généralement, les modèles sont les premiers composants d'un programme à être créés et doivent définir les protocoles devant être employés par les vues et les contrôleurs. Dans u n premier temps, il est possible de développer un modèle indépendamment de sa vue et de son contrôleur car l'environnement Smalltalk est suffisamment complet pour permettre de simuler les protocoles qui seraient utilisés par les vues et les contrôleurs.

Comme on peut le voir, les responsabilités communes à tous les modèles ne semblent pas tellement différentes de celles de n'importe quel autre type d'objet en Smalltalk. C'est pourquoi, n'importe quel objet peut constituer un modèle.

Toutes les classes du système sont organisées en hiérarchie (point 4.1) dont

la racine est la classe Object. Aussi, puisque toutes les classes héritent des propriétés (messages et variables) de leurs super-classes, chaque classe hérite des propriétés de la classe Object. C'est pourquoi le petit nombre de messages auxquels tout modèle doit pouvoir obéir a été implémenté dans cette classe. Ceci a également pour conséquence que le programmeur est libre de choisir son modèle dans la hiérarchie des classes du système Smalltalk. Ainsi, s'il juge qu'un string peut contenir l'information requise pour son programme - comme c'est le cas d'un éditeur de texte -, il décide que ce string forme un modèle. De la même façon, il peut créer une classe plus complexe, soit Dossier, et décider qu'une instance de cette classe forme un modèle pour son programme.

Comme tout objet Smalltalk, un modèle représente l'état et le comportement d'un objet du monde réel. Le détail de la représentation dépend habituellement de la nature de l'objet réel et du niveau de compréhension auquel il doit être compris, étant donné les objectifs du programme. Ainsi un simple nombre pourrait servir à modéliser le temps en enregistrant la température ou la force du vent. Par contre, on pourrait envisager un objet plus complexe pour modéliser un bureau et ses composants. En outre, un objet du monde réel peut se composer de nombreuses parties de type distinct. L'état d'un tel modèle serait composé de nombreuses parties, elles-mêmes étant des objets. Pour le problème qui nous concerne, une armoire devra être caractérisée notamment par des objets tiroirs, ceux-ci par les objets qu'ils contiennent, etc ...

Jusqu'à présent, nous avons surtout présenté un modèle comme une simple base de données. Toutefois, précisons qu'un modèle est également caractérisé, comme tout objet Smalltalk, par son interface contenant les messages auxquels il doit pouvoir répondre.

Deuxième étape: la conception de la vue.

a) La vue

La vue est le mécanisme qui permet de présenter à l'écran l'information fournie par un modèle. Elle constitue le mécanisme intermédiaire entre le modèle et les fenêtres. En Smalltalk, strictement tout est un objet (point 3.4.2.1), dès lors une vue est un objet et doit donc être une instance d'une classe du système. Suivant son type et ses fonctionnalités, une vue est une instance d'une des nombreuses classes décrivant des vues et s'insérant dans la hiérarchie du système. On y trouve par exemple les classes FormView, ListView, StandardSystemView, StringHolderView ou encore BrowserView, chacune d'entre elles définissant un type particulier de vue.

Chacune des classes que nous venons de citer, possède la classe **View** dans la chaîne de ses super-classes. Cette classe regroupe la déclaration des variables et des méthodes communes à toutes les vues. Chacune des classes décrivant un type particulier de vue représentent donc un raffinement de la classe View de laquelle elle hérite toutes les variables et méthodes nécessaires à son fonctionnement.

b) Les relations Vue-Modèle

La classe de la vue détermine comment afficher l'information provenant du

modèle. Chaque instance d'une vue connaît le modèle qu'elle présente. Si les données du modèle sont modifiées, la vue répercute les modifications à l'écran. La vue est donc dépendante du modèle. Cette relation de dépendance est implémentée grâce à la variable d'instance **model**, définie dans la classe View. La variable model permet de lier chaque vue à son modèle, elle constitue donc une référence explicite au modèle présenté par la vue. Par contre, les modèles ne peuvent se référer directement à leurs propres vues (figure 4.6). L'idée est de garder les modèles aussi simples que possible de manière à ce qu'ils ne sachent pas comment ils doivent s'afficher dans les fenêtres.

En outre, il est possible d'offrir différentes vues d'un même modèle, chaque vue déterminant une fenêtre particulière où s'affichent les informations. Ces vues référencent donc le même modèle. Les vues demandent à leur modèle respectif les informations à présenter à l'écran. La nature de ces demandes et la présentation varient de vue à vue. Si elles partagent le même modèle, elles recevront les mêmes informations. Si un modèle partagé subit une modification, les vues répercutent les changements à l'écran.

Idéalement, un modèle est tout à fait indépendant de la vue et ne se soucie pas de la manière dont est présentée l'information. Son rôle se borne alors à fournir les données à afficher demandées par sa (ses) vue(s).

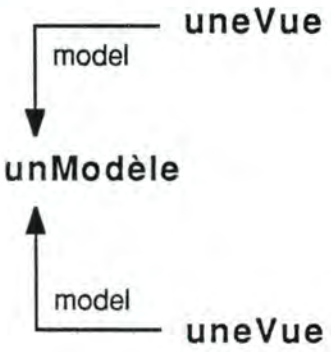


figure 4.6: Schématisation des relations entre un modèle et ses deux vues

Troisième étape: la conception du contrôleur

a) Le contrôleur

Le contrôleur se charge de gérer les communications entre l'utilisateur et les vues et/ou modèles. Il doit détecter les commandes et données introduites par l'utilisateur, les interpréter et déclencher une réponse adéquate. Ces données et commandes comprennent les mouvements de la souris, les pressions sur les boutons de la souris et du clavier.

Comme les vues, les contrôleurs sont des objets Smalltalk et sont donc des instances de classes qui les décrivent. On trouve, dans le système, différentes classes telles que MouseMenuController, StandardSystemController, ... qui définissent des contrôleurs particuliers. Chacune de ces classes possède la classe **Controller** dans la

chaîne de ses super-classes. C'est dans cette classe que sont définies les variables et méthodes communes à tous les contrôleurs.

b) Les relations Contrôleur-Vue et Contrôleur-Modèle

Chaque vue possède un contrôleur particulier chargé d'interpréter les données dans la fenêtre définie par la vue. Les boutons du clavier et de la souris prennent différentes significations dans les différentes fenêtres du simple fait que différents contrôleurs les interprètent.

- **controller**, variable définie dans la classe View, contient une référence explicite au contrôleur de la vue

- **view**, variable définie dans la classe Controller, contient une référence explicite à la vue du contrôleur.

Il faut remarquer, à nouveau, que le modèle ne possède aucune référence explicite vers le contrôleur tandis que le contrôleur possède une variable d'instance **model** contenant une référence vers le modèle.

La figure 4.7 présente de manière schématique les relations existants entre un modèle, deux de ses vues et deux de ses contrôleurs.

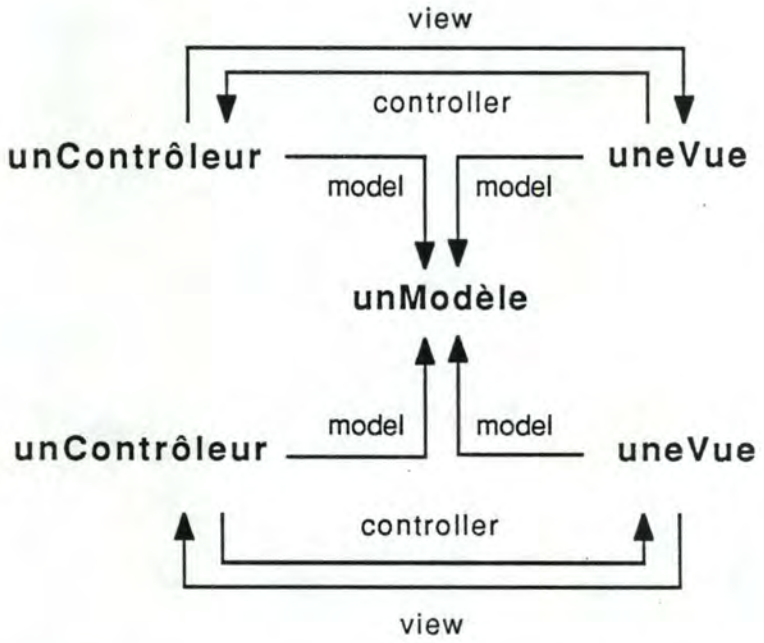


figure 4.7: Schématisation des relations entre un modèle, ses deux vues et ses deux contrôleurs

4.2.3 Conclusion

Le schéma de la figure 4.4 peut être adapté à la hiérarchie

Modèle-Vue-Contrôleur (figure 4.8). Dans ce dessin, les modèles forment la couche application, les vues et contrôleurs forment la couche présentation. A un modèle peut être associé plusieurs vues et à chaque vue est associée un contrôleur. Les vues et contrôleurs reposent sur les primitives graphiques présentes dans le système.

Grâce à cette architecture, des interfaces-utilisateurs très complexes peuvent être créées et utilisées. Des classes de base - View et controller - fournissent les structures de données et méthodes nécessaires à la création des vues et des contrôleurs. Couplé à cette architecture, le système Smalltalk devient un outil de développement de programmes très puissant. Cependant, un manque flagrant de documentation rend très malaisée la manipulation des classes View et Controller ainsi que de leurs sous-classes. Le lecteur interessé pourra trouver quelques informations dans [BCOX-86], [CUNN-85], [DIED-87] et dans le manuel d'utilisation du Smalltalk sur les stations de travail qui l'implémentent.

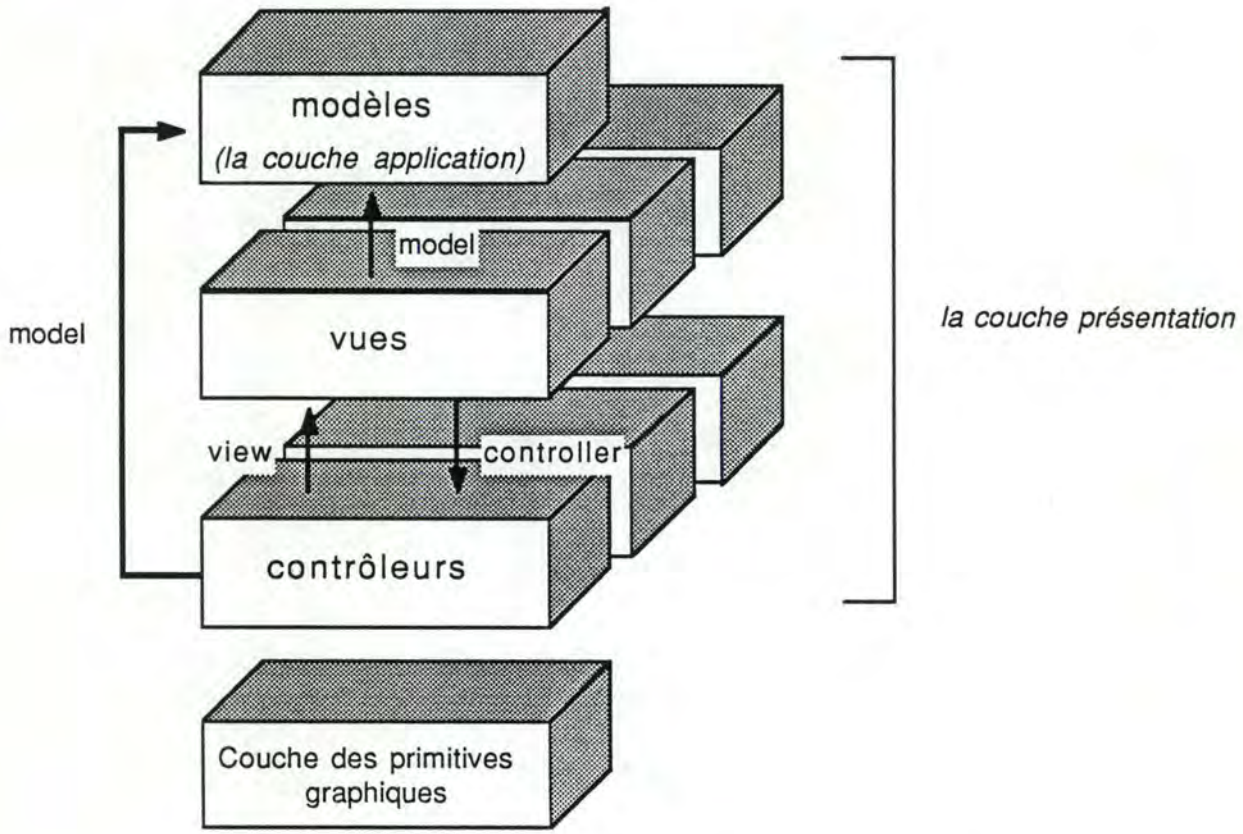


figure 4.8: L'architecture M.V.C. du système Smalltalk-80

4.3 Les classes Workspace et Browser

Les quelques classes que nous avons exposées au point 4.1 constitue un tout petit sous-ensemble des classes de base du système Smalltalk. A côté de celles-ci, on trouve également une multitude d'autres classes dont les fonctionnalités sont des plus diverses. Notre objectif dans ce point n'est pas du tout d'analyser chacune de ces classes, mais d'en examiner deux qui nous seront utiles pour la suite de notre mémoire. Nous verrons également que ces classes peuvent constituer des exemples

d'application de l'architecture M.V.C..

4.3.1 La classe Workspace

Au point 3.4.5, lorsque nous avons exposé les principes de base de l'utilisation de l'environnement de programmation Smalltalk, nous avons vu que l'utilisateur dispose de fenêtres spéciales appelées Workspace. Ces fenêtres sont notamment destinées à recevoir du texte qui sera sélectionné et exécuté au moyen de la souris.

Comme tout outil de ce genre, le workspace fait appel à l'architecture Modèle-Vue-Contrôleur décrite au point 4.2. Il est donc composé de trois éléments: un modèle, une vue et un contrôleur.

Le modèle est décrit dans la classe Workspace dont la super-classe est la classe StringHolder. Elle définit une structure de données permettant notamment de conserver le texte présenté sous forme de string, la position du curseur ou encore le texte éventuellement sélectionné dans les vues auxquelles sont associées les instances. La classe Workspace fournit également les messages de manipulation de ces données et de gestion de la coopération avec les vues et contrôleurs des workspaces.

La vue est quant à elle décrite par la classe WorkspaceView dont la super-classe est la classe StringHolderView. Cette classe possède, dans son protocole, les messages utilisés pour l'affichage de texte mais aussi les messages destinés à répondre aux commandes introduites par l'utilisateur par l'intermédiaire du contrôleur. Les instances de la classe WorkspaceView héritent des variables d'instance *model* et *controller* déclarées dans la classe View. Elles possèdent aussi une référence vers les modèles et les contrôleurs auxquels ils sont associés.

La classe StringHolderController fournit la description des contrôleurs associés aux instances de la classe WorkspaceView. On y trouve, entre autres, la définition des Pop-Up menus et des méthodes déclenchées par le choix de chacune des options de ceux-ci. Les instances de la classe StringHolderController héritent des variables d'instance *model* et *view* déclarées dans la classe Controller. Elles possèdent alors une référence explicite vers les modèles et les vues auxquels elles sont associées.

A la figure 4.9 apparaissent les relations entre les composants que l'on vient de décrire.

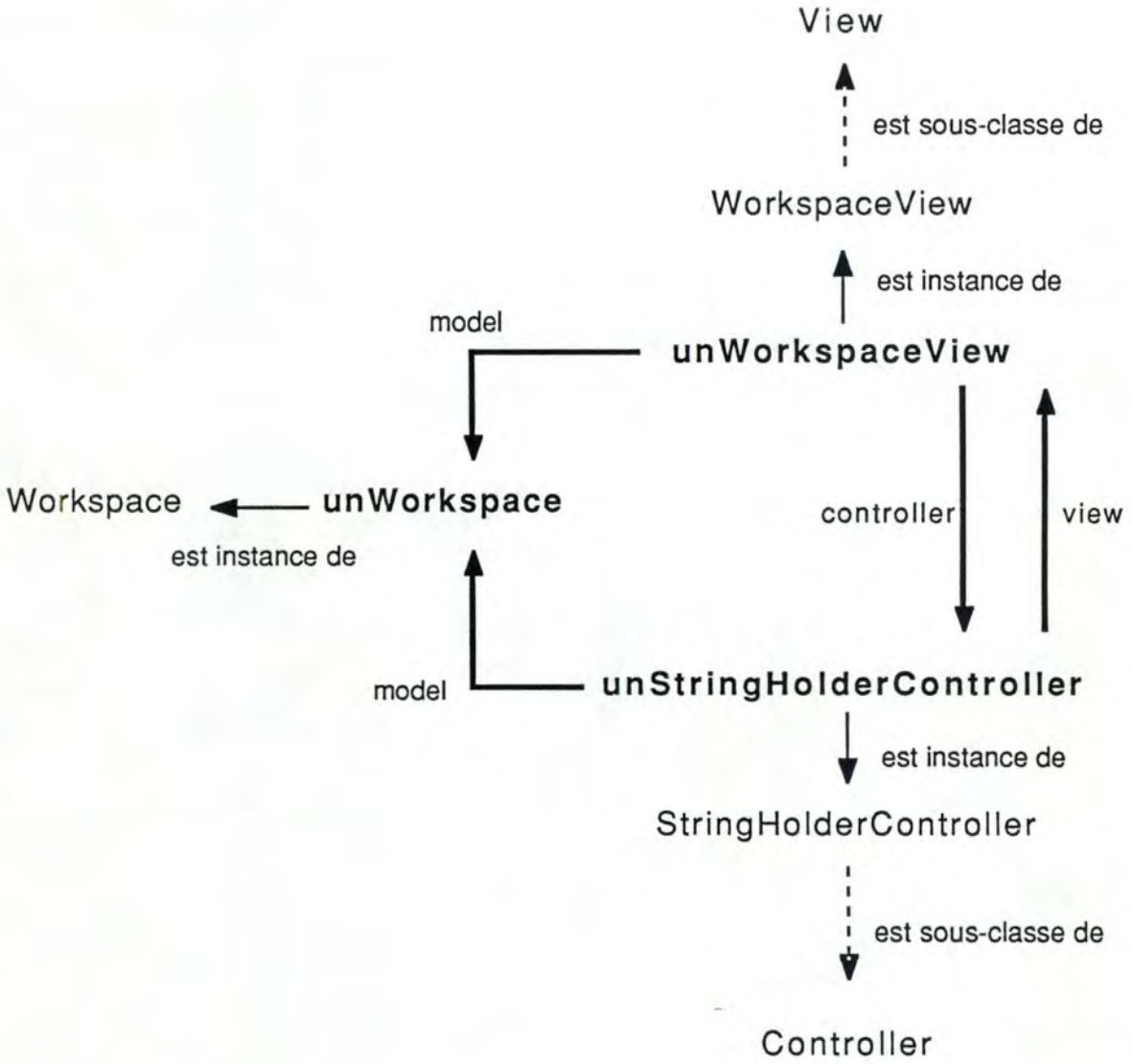


figure 4.9: Les relations existants entre les composants d'un workspace

4.3.2 La classe Browser

Dans l'environnement Smalltalk, outre les fenêtres Workspace, on trouve également les fenêtres System Browser (point 3.4.5). Ces fenêtres permettent de naviguer à travers la hiérarchie des classes du système, de modifier, consulter ou rajouter de nouvelles classes et méthodes. Elles sont composées de plusieurs autres fenêtres auxquelles sont associées des Pop-Up menus spécifiques. Encore une fois, cet outil utilise l'architecture M.V.C..

Le modèle d'un tel outil est décrit dans la classe Browser. On y trouve tous les messages pour obtenir les informations à afficher dans les différentes fenêtres. Ces informations sont les listes des catégories (point 3.4.5), de classes et de messages ainsi que les définitions des classes et des méthodes. C'est également le modèle qui se charge d'insérer les nouvelles classes et les nouveaux messages dans le système.

La classe BrowserView décrit quant à elle la vue d'un System Browser. Grâce

aux messages de cette classe, la vue se charge d'ouvrir les différentes fenêtres composant l'outil et d'afficher les données nécessaires. Chacune de ces fenêtres est décrite dans une classe particulière et est associée à un contrôleur spécifique.

Dans le chapitre suivant, nous allons, sur base des informations données dans le chapitre 4, exposer les grandes lignes de l'architecture de notre logiciel.

CHAPITRE 5

L'ARCHITECTURE DE NOTRE LOGICIEL

CHAPITRE 5: L'ARCHITECTURE DU LOGICIEL

Ce chapitre est destiné à la description de notre implémentation de l'animation graphique des tâches de bureau. Nous précisons immédiatement que cette description est la plus intuitive possible, de manière à donner au lecteur une vision globale de l'implémentation que nous proposons, et cela sans rentrer dans des détails techniques inutiles pour le moment. Le développement complet de l'implémentation ainsi que notre démarche de conception seront détaillés dans les deux chapitres suivants.

5.1 L'architecture de notre logiciel

Afin de présenter des simulations graphiques intéressantes et agréables, et comme c'est souvent le cas pour la plupart des applications écrites dans le système Smalltalk-80, notre programme a fait largement appel aux outils de création d'interface-utilisateur disponibles. Ceux-ci sont organisés selon une architecture en trois couches, appelées modèle, vue et contrôleur, et déjà exposées au chapitre précédent (point 4.2).

Pour établir notre propre architecture à partir de ces trois couches, nous avons dû créer différentes classes, chaque classe faisant partie de la couche modèle, vue ou contrôleur de notre logiciel.

Dans les paragraphes suivants, nous allons déterminer les classes dont les instances servent de modèle, de vue ou de contrôleur pour notre programme, nous montrerons la hiérarche de classes de notre logiciel et nous exposerons brièvement les variables d'instance que nous avons définies.

5.1.1 La couche modèle

L'objectif du mémoire consiste à réaliser un outil de simulation de tâches de bureau. Puisqu'un modèle représente les données d'un logiciel et les opérations qui peuvent être exécutées sur ces données (point 4.2.2.2), nous avons été amenés à considérer que les objets du bureau tant au niveau technologique (point 2.3.2) qu'informationnel (point 2.3.1.1) forment le modèle de notre logiciel.

Rappelons (cfr chapitre 2) qu'il s'agit:

- des *objets informationnels* qui peuvent se partager en objets informationnels *élémentaires* ou *structurants*. Ces deux ensembles d'objets peuvent eux-mêmes se répartir respectivement en *message*, *document* et *formulaire* ou *fichier*, *dossier* et *pile*

- des *objets de rangement* parmi lesquels on trouve les *armoires*, les *tiroirs*, les *boîtes*, les *boîtes à archives*, les *fardes* et les *étagères*

- des *objets d'interface* c'est-à-dire les *boîtes IN* et les *boîtes OUT*, la *poubelle* et le *téléphone*
- des *outils de travail* qui peuvent être des *tables de travail* ou des

photocopieuses .

Sur base des définitions données au chapitre 2, nous avons déterminé des classes d'objets informationnels, de rangement, d'interface et de travail. Ces classes regroupent les objets du monde réel qui ont des comportements communs et servent de modèle pour la création des objets qui les représentent.

Nous avons donc créé les classes suivantes:

- **Information** qui est chargée de décrire la structure de données et les protocoles propres à tous les objets informationnels

- **InfoElémentaire** qui regroupe les propriétés des objets informationnels non décomposables en d'autres objets informationnels

- **InfoStructurant** qui regroupe les propriétés des objets informationnels décomposables en objets informationnels élémentaires et/ou structurants

- **Rangement** qui est chargée de décrire la structure de données et les protocoles propres à tous les objets de rangement

- **Interface** qui est chargée de décrire la structure de données et les protocoles propres à tous les objets d'interface

- **Travail** qui est chargée de décrire la structure de données et les protocoles propres à tous les outils de travail

Puisque chaque objet informationnel, de rangement, d'interface ou de travail possède ses propres caractéristiques, il est normal de leur associer une classe particulière. Nous considérons donc également les classes:

- **Msge**
- **Formulaire**
- **Document**

- **Dossier**
- **Fichier**
- **Pile**

- **Armoire**
- **Etagère**
- **Tiroir**
- **Farde**
- **Boîte**
- **BoîteArchive**

- **Téléphone**
- **Poubelle**
- **BoîteIN**
- **BoîteOut**

- **TableTravail**
- **Photocopieuse.**

Après avoir défini les nouvelles classes, nous les avons organisées de manière hiérarchique de façon à profiter au maximum du principe d'héritage propre aux langages orientés objet et à Smalltalk (point 3.4.3). Cela permet d'éviter des redondances inutiles de code.

Lorsqu'on examine les différentes classes d'objets que l'on vient de passer en revue, on peut aisément se convaincre qu'il est possible d'intégrer un niveau d'abstraction (point 3.5.1) et de généralisation (point 3.5.2) supplémentaire. En effet, lors d'une analyse de bureau, tous les objets répertoriés appartiennent au même bureau et ont des comportements communs. Ceci nous amène à ajouter une nouvelle classe appelée **Composant**, chargée de regrouper les structures de données et les méthodes communes et nécessaires à tout objet de bureau. Cette nouvelle classe a été définie comme sous-classe de la classe **Object** (point 4.1.1), pour pouvoir insérer notre hiérarchie de classes dans la hiérarchie du système Smalltalk-80 (cfr chapitre 4). La classe Object a été choisie comme super-classe de la classe Composant car Object rassemble, rappelons-le, toutes les caractéristiques de base nécessaires à tous les modèles. Nos propres modèles pourront donc en hériter directement sans hériter d'autres données inutiles à notre logiciel.

Les classes que nous avons dégagées jusqu'à présent constituent les classes de base de notre logiciel. Cependant, pour que ce noyau de base soit complet, il faut y ajouter la classe **Bureau** qui décrit les structures de données permettant de conserver l'ensemble des objets informationnels, de rangement, d'interface et de travail se trouvant dans un bureau donné. La classe Bureau a Object pour super-classe car elle n'a besoin que des variables et méthodes de base nécessaires à tous les modèles.

La hiérarchie des classes se présente donc comme suit (figure 5.1):

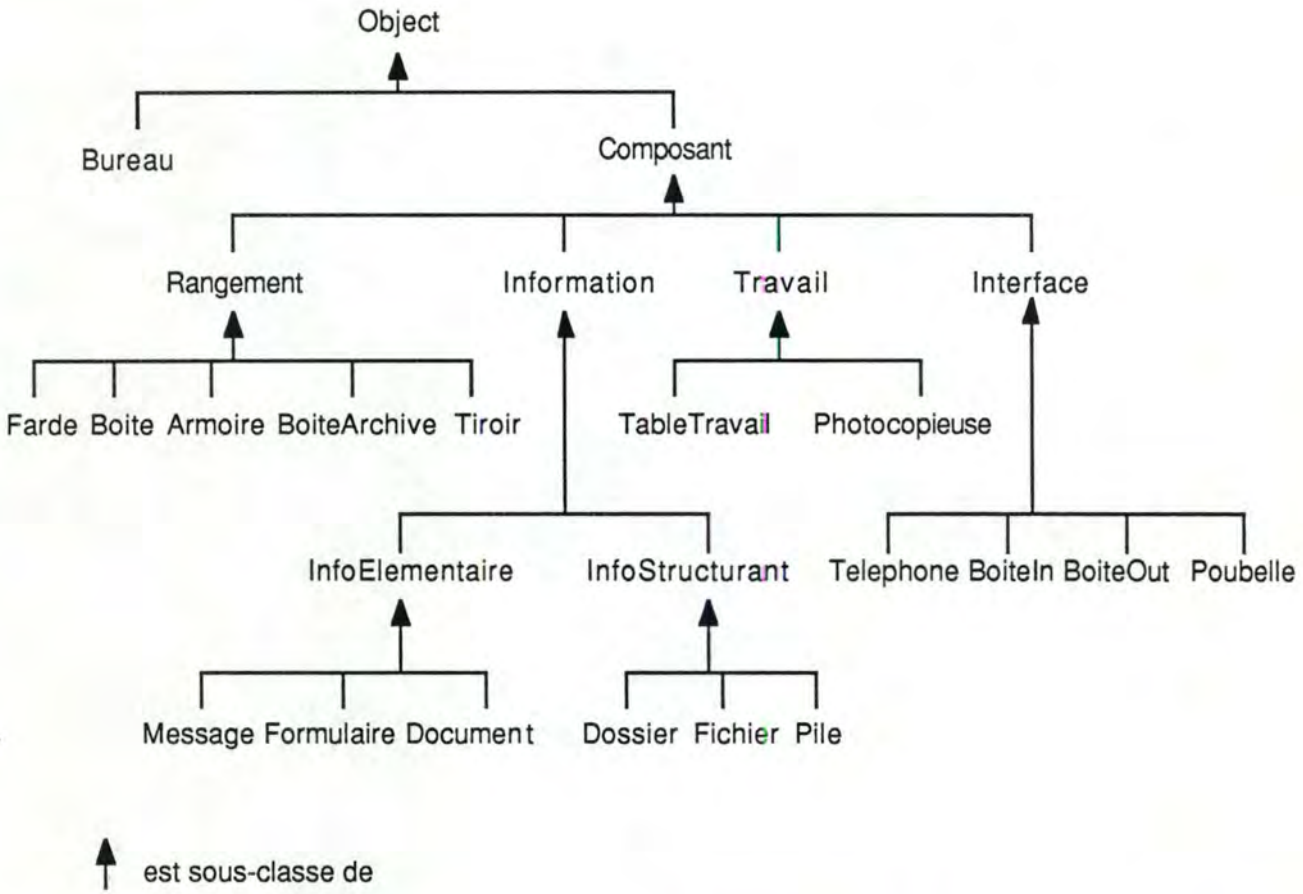


figure 5.1: Hiérarchie des classes de notre logiciel

5.1.1.1 Les variables d'instance

Tous les objets sont qualifiés grâce à des attributs (point 2.5) permettant de les différencier. Les informations représentant ces attributs sont rangées dans un ensemble de variables d'instance. Rappelons que toutes les instances d'une classe ont les mêmes variables d'instance mais que, d'une instance à l'autre, les valeurs de ces variables peuvent être différentes.

Puisque tous les objets informationnels sont caractérisés par les mêmes attributs (*nom*, *type*, *numéro de copie*, *état*, *échéance*, *ordre de classement*, *état du contenu*), nous associerons les variables d'instance à la classe Information, toutes les sous-classes d'Information en héritant automatiquement.

De la même façon, puisque tous les objets informationnels, de rangement, d'interface et de travail sont caractérisés par le même attribut *nom*, nous l'associerons, sous forme d'une variable d'instance, à la classe Composant; toutes les sous-classes de Composant en héritant automatiquement.

Il est à noter que nous n'avons pas repris ici toutes les variables d'instance que nous avons dû définir en cours d'implémentation; ce chapitre ne sert en fait qu'à donner une vision globale de la structure de notre logiciel. Ces autres variables seront définies ultérieurement.

Nous arrivons donc à représenter graphiquement l'architecture de notre logiciel de la sorte: (figure 5.2).

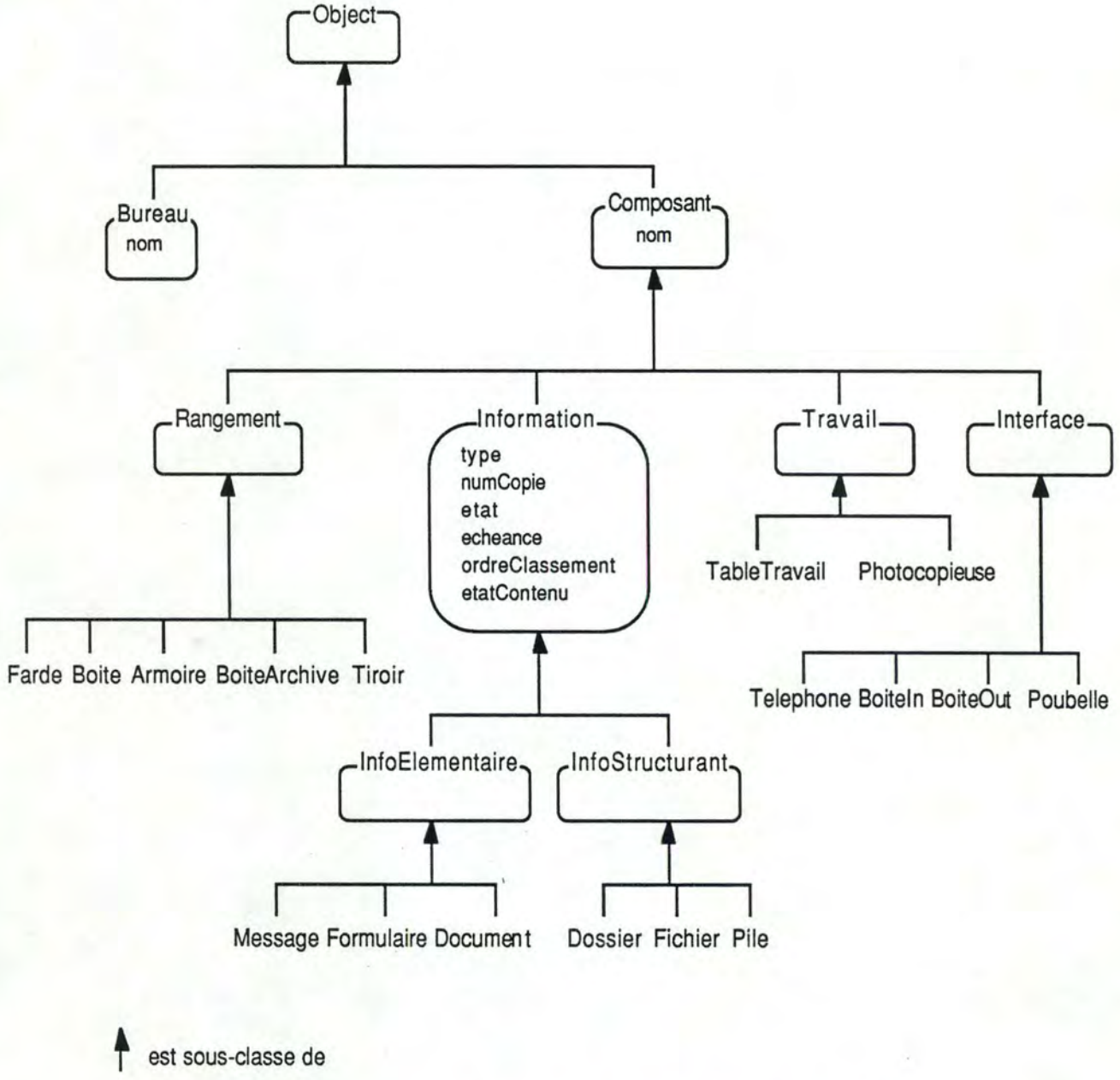


figure 5.2: La hiérarchie des classes de modèles de notre logiciel

5.1.2 Les couches Vue et Contrôleur

Très brièvement, nous allons présenter maintenant les nouvelles classes permettant d'afficher à l'écran le bureau ainsi que ses composants et d'exécuter les simulations des tâches.

Il s'agit des classes:

- **BureauView** chargée de définir la vue dans laquelle s'exécute la simulation

des tâches.

- **ComposantView** chargée de définir les vues dans lesquelles s'affichent les composants du bureau, c'est-à-dire les objets d'interface, de travail, les objets de rangement ne se trouvant dans aucun autre objet de rangement et les objets informationnels affichés uniquement lors d'une simulation.

- **BureauController** permettant à l'analyste de communiquer avec le bureau.

- **ComposantController** permettant à l'analyste de déplacer les composants d'un bureau, de les identifier et d'obtenir un répertoire indiquant leur contenu.

Les classes BureauView et ComposantView ont la classe **View** (point 4.2.2.2) pour super-classe car elles ne se basent que sur les variables et méthodes de base communes à toutes les vues.

Les classes BureauController et ComposantController ont la classe **MouseMenuController** (point 4.2.2.2) pour super-classe car elles ne se basent que sur les variables et méthodes de base communes à tous les contrôleurs.

En conclusion, la figure 5.3 présente un récapitulatif des différentes classes que nous venons d'ajouter.

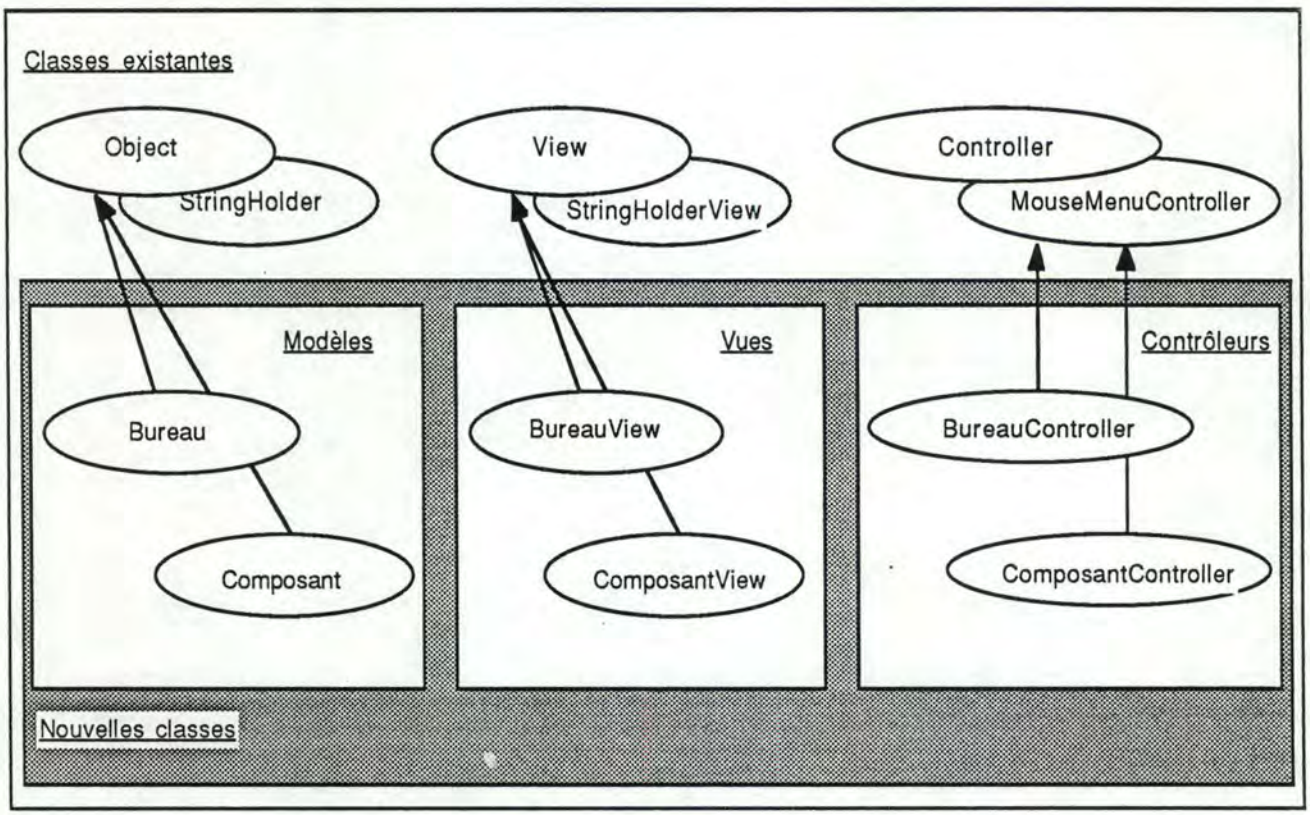


figure 5.3: L'architecture modèle, vue, contrôleur de notre logiciel

CHAPITRE 6

LES MODELES DE NOTRE LOGICIEL

CHAPITRE 6: LES MODELES DE NOTRE LOGICIEL

Un modèle représente les données d'un programme et les opérations qui peuvent être exécutées sur ces données (point 4.2.2.2). Généralement, les modèles sont les premiers composants d'un programme à être créés.

Dans ce chapitre, nous allons déterminer les classes dont les instances serviront de modèles dans notre programme. Pour ce faire, nous travaillerons de manière progressive en partant d'une première hiérarchie de base que nous enrichirons petit à petit. Nous nous baserons sur les principes d'abstraction et de généralisation exposés au chapitre trois (points 3.5.1 et 3.5.2).

6.1 La détermination des modèles de base

L'objectif de ce mémoire étant de réaliser un outil de simulation de tâches de bureau, nous sommes amenés à considérer des objets consistant en de l'information manipulée par une tâche et en de la technologie utilisée par cette même tâche pour s'exécuter.

Nous allons examiner chacun des types d'objets en procédant comme suit: pour chaque type, nous déterminerons les classes dont les instances nous serviront de modèles; ensuite, nous organiserons ces classes en hiérarchie, nous définirons les variables d'instance et éventuellement les variables de classe devant déterminer l'espace mémoire privé des objets, et enfin nous déterminerons les messages formant les interfaces.

Les classes que nous aurons dégagées constitueront les classes de base de notre programme. En effet, elles définissent les éléments fondamentaux que l'on peut distinguer après une première analyse des spécifications du problème. Par la suite, nous verrons que d'autres classes ont été nécessaires pour construire notre programme.

6.1.1 Les objets informationnels

6.1.1.1 Les classes

Sur base des définitions données au chapitre 2, nous allons déterminer les classes d'objets informationnels que nous avons utilisées.

La première classe que l'on nomme **Information** sera chargée de décrire la structure de données (variables d'instance et de classe) et les protocoles propres à tous les objets informationnels quel que soit leur type.

En analysant les définitions des objets informationnels (point 2.2.1), on s'aperçoit qu'ils peuvent adopter des comportements différents selon qu'ils sont élémentaires ou structurants. C'est pourquoi deux nouvelles classes ont été créées:

* **InfoElementaire** qui regroupe toutes les propriétés des objets informationnels qui ne peuvent pas être décomposés pour donner

d'autres objets informationnels,

* **InfoStructurant** qui rassemble les propriétés des objets informationnels pouvant être composés d'autres objets informationnels élémentaires et/ou structurants.

Les objets informationnels élémentaires et les objets informationnels structurants peuvent chacun se décomposer en trois types d'objets différents. Puisque chaque objet possède ses propres caractéristiques, il est utile de leur associer une classe particulière.

En résumé, on obtient le noyau de classes de base suivant: **Information, InfoElementaire, InfoStructurant, Dossier, Fichier, Pile, Message, Formulaire, Document.**

6.1.1.2 La hiérarchie des classes

Les définitions des objets informationnels suivent un mécanisme de spécialisation progressive, en partant de la generalité Information pour arriver à des objets tels que Fichier, Message ou encore Document. Ces définitions offrent, en quelque sorte, une nomenclature des objets informationnels qui correspond à une organisation hiérarchique. La classe Information constitue le niveau d'abstraction et de généralisation le plus élevé et forme la racine de cette "mini-hiérarchie". A l'opposé, les classes Dossier, Fichier, Pile, Message, Formulaire et Document formeront le niveau d'abstraction le plus faible et constitueront le dernier niveau de la hiérarchie.

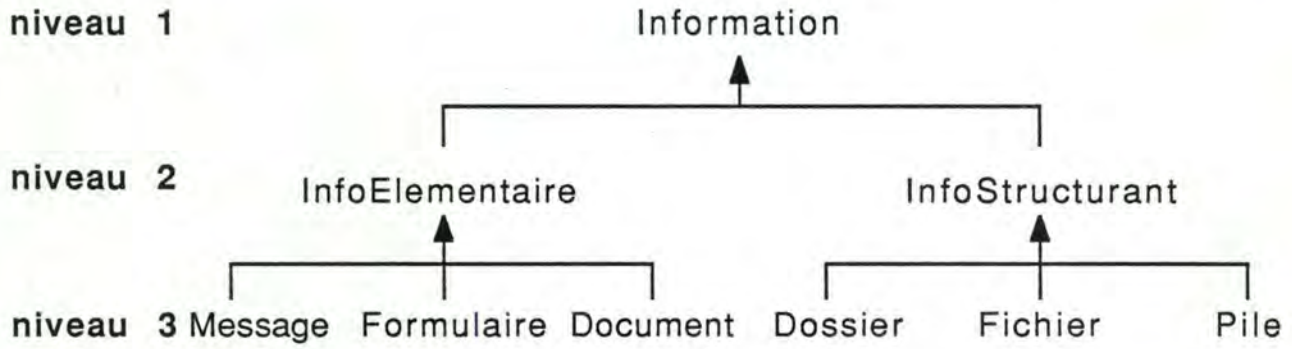


figure 6.1: La hiérarchie des objets informationnels

On obtient l'arbre hiérarchique de la figure 6.1 où les classes de niveau 3 représentent le niveau de spécialisation le plus élevé. Elles regroupent uniquement les spécificités propres à chaque type d'objets informationnels, que ce soit au niveau des variables d'instance et de classe ou au niveau des méthodes. En outre, ces classes bénéficieront, grâce à l'héritage, de toutes les propriétés de leurs super-classes, c'est-à-dire des propriétés des objets informationnels élémentaires ou structurants et récursivement de tous les objets informationnels en général.

Cela permet de faciliter l'ajout de nouvelles classes d'objets de niveau 3, et éventuellement de niveau supérieur, qui hériteront automatiquement de toutes les

propriétés des objets informationnels. On évite ainsi toute reprogrammation et duplication inutiles de code et de variables.

6.1.1.3 Structuration de l'état d'un objet informationnel

Tous les objets informationnels sont qualifiés grâce à un ensemble d'attributs (point 2.4.3.1) qui seront rangés dans des variables d'instance.

Puisque tous les objets informationnels, quelle que soit la classe à laquelle ils appartiennent, sont caractérisés par les mêmes attributs, plutôt que d'associer les mêmes variables d'instance à chaque classe de niveau 3 (figure 6.1), nous passons à un niveau de généralisation et d'abstraction plus élevé. Les variables d'instance sont dès lors associées à la classe Information, toutes les sous-classes en héritant automatiquement.

Les variables d'instance permettant de caractériser un objet informationnel sont:

* **nom**

  - la variable nom spécifie le nom de l'objet informationnel. Sa valeur est déterminée par l'analyste lors de la description de l'environnement du bureau.
  - la valeur de la variable est une instance de la classe Symbol (point 4.1.2.2).

* **type**

  - la variable type permet de répertorier, en types, les objets informationnels d'une même classe. Une sous-classification est introduite pour permettre à l'analyste de décrire des sous-classes parmi les objets informationnels de son bureau. Sa valeur est déterminée par l'analyste lors de la description de l'environnement du bureau.
  - sa valeur est une instance de la classe Symbol (point 4.1.2.2).

* **numCopie**

  - la variable numCopie spécifie le numéro de copie de l'objet informationnel. Par convention, un numéro nul indique qu'il s'agit d'un original.
  - la valeur de la variable est une instance de la classe SmallInteger (point 4.1.2.1).

* **état**

  - la variable état permet de connaître l'état d'avancement de l'objet informationnel dans son cycle de vie. Sa valeur initiale est donnée par l'analyste et peut changer en cours de tâche.
  - la valeur de cette variable est une instance de la classe Symbol (point 4.1.2.2).

* **échéance**

  - la variable échéance détermine la date à laquelle un objet informationnel doit être traité au plus tard. Le système ne gère pas cette date d'échéance.
  - la valeur de cette variable est une instance de la classe String (point 4.1.2.2) de la forme 'MM/JJ/AAAA'.

* **ordreClassement**

  - la variable ordreClassement détermine l'ordre dans lequel doivent se trouver les constituants éventuels de l'objet informationnel. Pour la pile, l'ordre #LIFO est imposé automatiquement et un objet informationnel élémentaire se voit attribuer l'ordre #AUCUN par le système.
  - le domaine de valeur de la variable est l'ensemble des instances suivantes de la classe Symbol (point 4.1.2.2):

    # LIFO
    # FIFO
    # ALPHABETIQUE
    # ALPHABETIQUEINVERSE
    # AUCUN

* **étatContenu**

  - la variable étatContenu indique si l'objet informationnel est vide ou pas. Un objet informationnel est vide s'il ne possède aucun constituant. lors de la création de l'objet informationnel, la valeur #VIDE est affectée à la variable par le système, ce dernier en assurant automatiquement la gestion.
  - le domaine de valeur de la variable est l'ensemble des instances suivantes de la classe Symbol (point 4.1.2.2):

    #VIDE,
    #NONVIDE.

La hiérachie de la figure 6.1 peut dès lors être complétée en ajoutant les variables que nous avons définies.

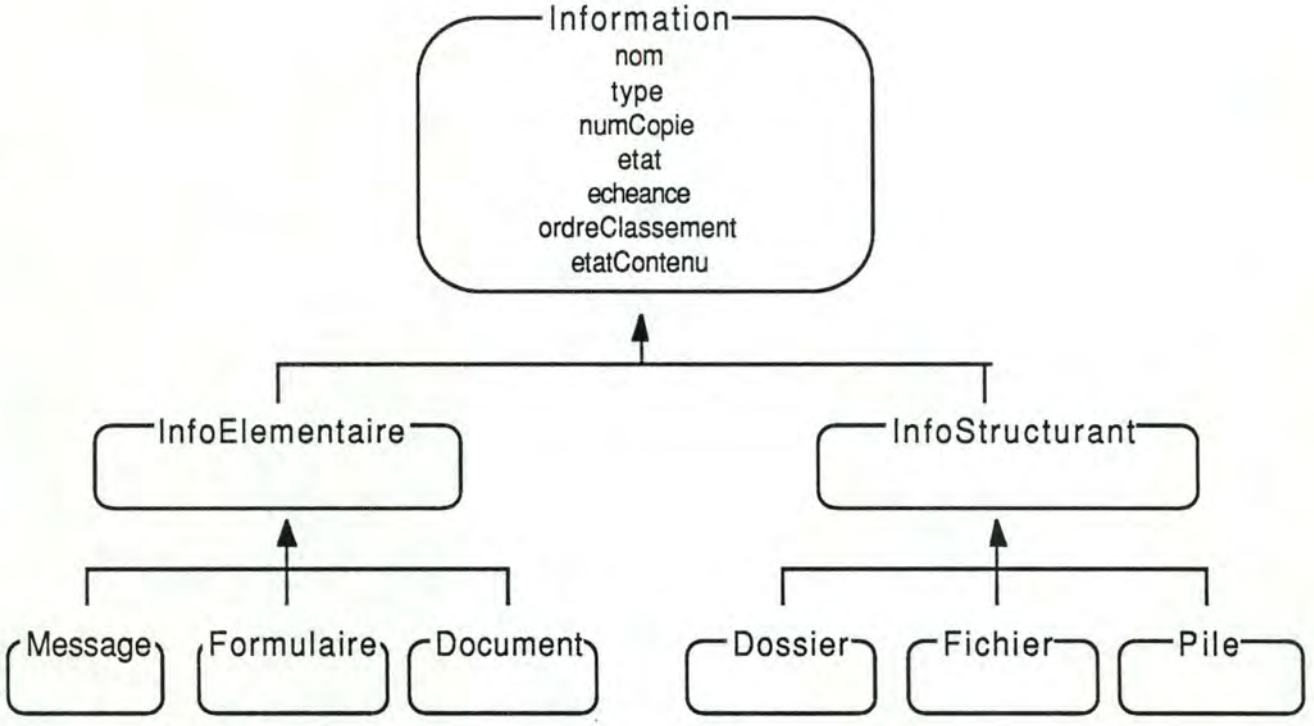


figure 6.2: Hiérarchie des objets informationnels et de leurs variables d'instance

Comme on peut le constater, tout objet informationnel hérite, quelle que soit sa classe, de l'ensemble des variables précitées. L'utilisateur peut donc leur affecter la valeur qu'il désire.

Toutefois, certaines de ces variables ne sont significatives et obligatoires que pour certains objets et falcultatives pour d'autres. Comme aucun contrôle n'est réalisé à ce niveau, ils devront donc l'être au niveau d'une couche supérieure permettant de gérer cette caractéristique.

Si l'analyste ne désire pas spécifier de valeur pour une variable, il lui affecte la valeur **nil** qui correspond à la valeur de tout objet Smalltalk non encore initialisé.

Une façon d'introduire un contrôle, au niveau des attributs essentiellement significatifs pour certaines classes d'objets, serait de redescendre les variables d'instance les représentant au niveau des classes. Toutefois, par souci de généralisation la plus élevée possible, nous avons préféré les associer toutes à la classe Information.

6.1.2 Les objets de rangement

6.1.2.1 Les classes

En partant des définitions données au chapitre 2, on aboutit à une première classe appelée **Rangement**. Cette classe est chargée de rassembler les variables d'instance et les méthodes pouvant être utilisées par tout objet de Rangement.

Ensuite, d'autres classes ont été définies: les classes Armoire, Tiroir, Boîte, Etagere, Farde, BoiteArchive. Chacune de ces classes correspond à un type d'objets

de rangement du même nom et permet d'apporter des spécificités quant au comportement des objets d'une classe par rapport à ceux d'une autre classe.

On a donc la liste des classes suivantes: **Rangement, Armoire, Tiroir, Etagere, Boîte, Farde, BoîteArchive.**

6.1.2.2 La hiérarchie des classes

D'après la définition des objets et la découpe en classes, il apparait que la classe Rangement représente le niveau d'abstraction et de généralisation le plus élevé, alors que les autres classes constituent le niveau de spécialisation le plus fort. Toutefois, ces dernières classes doivent pouvoir hériter des variables d'instance et des méthodes propres à tous les objets de rangement. Nous organisons donc l'ensemble des classes de la manière présentée à la figure 6.3 où Rangement est la super-classe des autres classes.

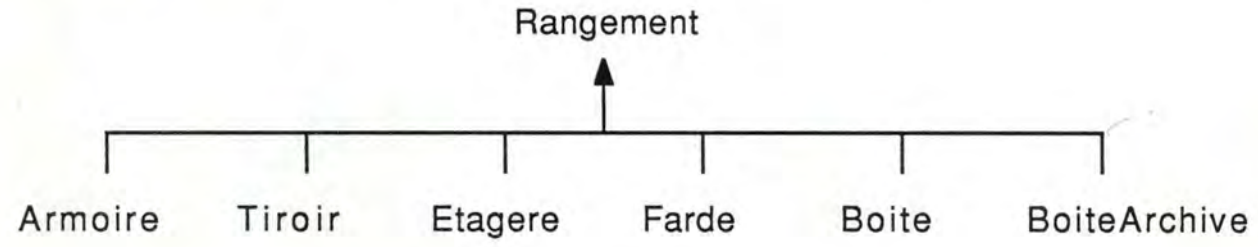


figure 6.3: Hiérarchie des objets de rangement

6.1.2.3 Structuration de l'état d'un objet de rangement

Puisque tout objet de rangement, quelle que soit sa classe, possède un nom, nous implémentons cet attribut sous forme d'une variable d'instance **nom** associée à la classe Rangement. La valeur de cette variable est une instance de la classe Symbol et sera donnée par l'analyste lors de la description de l'environnement de bureau.

La hiérarchie de la figure 6.3 (figure 6.4) peut être complétée en y ajoutant la variable que nous avons définie.

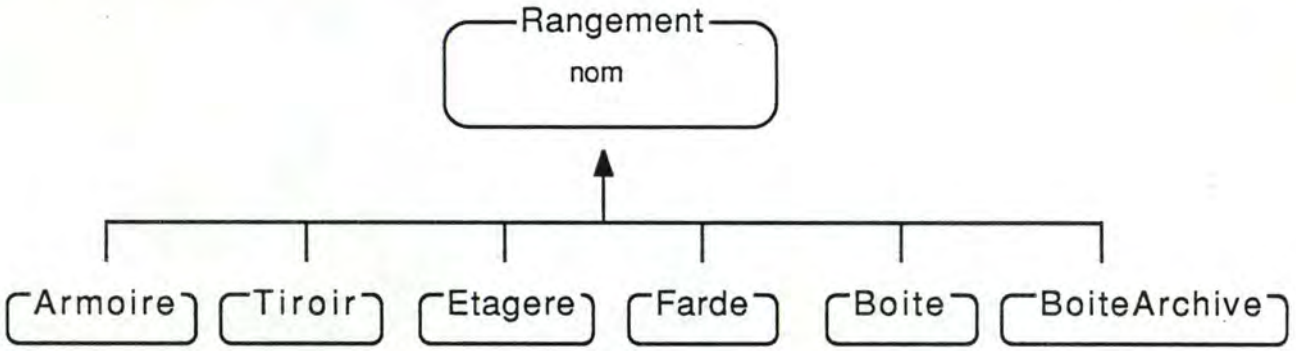


figure 6.4: Hiérarchie des objets de rangement et leur variable d'instance

6.1.3 Les objets d'interface

Pour les objets d'interface et de travail, la démarche suivie est identique. C'est pourquoi, nous nous contenterons de donner la liste des classes, leur hiérarchie et leurs variable d'instance, ainsi qu'un bref mot d'explication.

6.1.3.1 Les classes

Nous définissons les classes suivantes:

- **Interface** qui regroupe les caractéristiques propres à tous les objets d'interface.
- **BoiteIn**
- **BoiteOut**
- **Telephone**
- **Poubelle**

Chacune des quatre dernières classes regroupe les spécificités propres à chaque type d'objet d'interface particulier.

6.1.3.2 La hiérachie des classes

Les classes que l'on vient de recenser peuvent être organisées de la manière suivante:

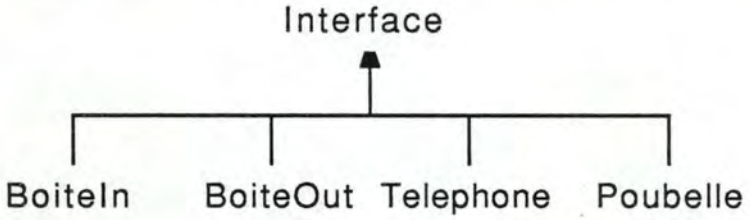


figure 6.5: Hiérarchie des objets d'interface

où Interface est la super-classe des autres classes, celles-ci héritant alors de toutes les propriétés communes à tous les objets d'interface.

6.1.3.3 Structuration de l'état d'un objet d'interface

Le nom de l'objet d'interface est le seul attribut utilisé. Ce nom, donné par l'analyste, est une instance de la classe Symbol.

La variable d'instance représentant cet attribut complète la hiérarchie de la figure 6.5 (figure 6.6)

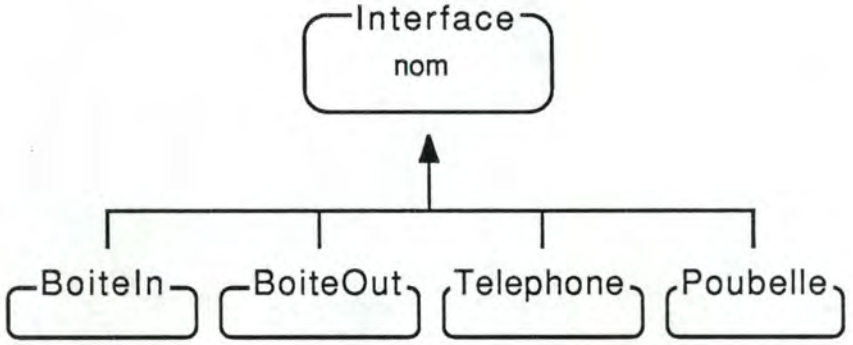


figure 6.6: Hiérarchie des objets d'interface et de leur variable d'instance

6.1.4 Les objets de travail

6.1.4.1 Les classes

Les classes suivantes sont utilisées:

- **Travail** qui regroupe les caractéristiques propres à tous les objets de travail.
- **TableTravail**
- **Photocopieuse**

Chacune des deux dernières classes regroupe les spécificités propres à chaque type d'objet de travail particulier.

6.1.4.2 La hiérarchie des classes

Les classes que l'on vient de recenser peuvent être organisées de la manière présentée à la figure 6.7 où Travail est la super-classe des deux autres classes, celles-ci héritant de toutes les propriétés communes aux objets de travail.

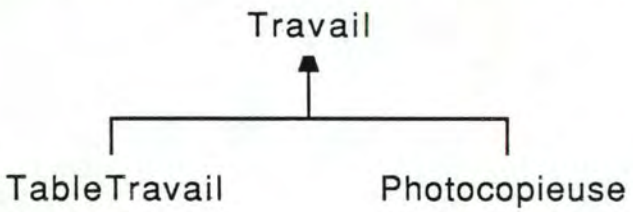


figure 6.7: Hiérarchie des objets de travail

6.1.4.3 Structuration de l'état d'un objet de travail

Le seul attribut que l'on utilisera pour l'instant, sera le nom de l'objet de travail. Ce nom sera donné par l'analyste et devra être une instance de la classe Symbol.

La variable d'instance représentant cet attribut complète la hiérarchie de la

figure 6.7 (figure 6.8).

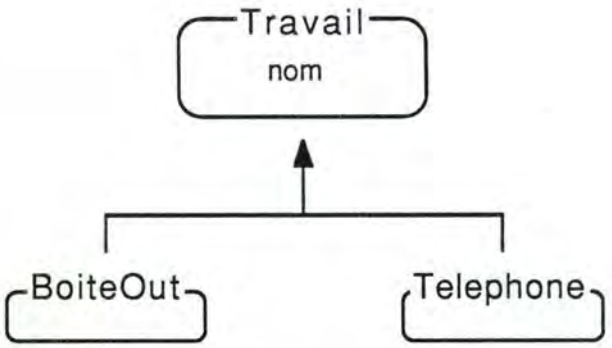


figure 6.8: Hiérarchie des objets de travail et de leur variable d'instance

Au sein des hiérarchies de classes que nous venons de passer en revue, seules les classes Information, Rangement, Interface et Travail sont intangibles. Toutes les autres classes d'objet de notre programme peuvent être enlevées de notre hiérarchie.

6.2 La classe Composant

Lors de l'analyse d'un bureau, tous les objets recensés, qu'ils soient informationnels, de rangement, d'interface et de travail, ont un point commun: celui d'appartenir au même bureau. Ceci nous laisse supposer que tous les objets de bureau ont des comportements communs, quelle que soit leur classe, c'est-à-dire qu'ils sont censés pouvoir répondre à un ensemble de messages de même type assurant par exemple la coopération avec le bureau auquel ils appartiennent. Il y a donc des risques de duplications inutiles au niveau du code si les méthodes sont implémentées dans chacune des classes Information, Rangement, Interface et Travail.

En outre, la détermination des variables d'instance propres à chacune des super-classes Information, Rangement, Interface et Travail, fait apparaître que chacune d'entre elles possède la variable *nom* . De plus, lors de l'implémentation du logiciel, des variables supplémentaires et des méthodes devront vraisemblablement être ajoutées de manière à gérer la coopération dont on vient de parler. Ces variables devront être reproduites dans chacune des super-classes.

Cette argumentation nous pousse à ajouter une nouvelle classe appelée **Composant**, chargée de regrouper les structures de données et méthodes communes et nécessaires à tout objet de bureau. Cette classe sera la super-classe des classes Information, Rangement, Travail, et Interface. De cette manière, toutes les classes définies au point 6.1 peuvent hériter des propriétés communes à tout objet de bureau en évitant les duplications. La classe **Object** (point 4.1.1) a été choisie comme super-classe de la classe Composant. Cette classe Object, rappelons-le, rassemble toutes les caractéristiques de base nécessaires à tous les modèles.

L'arbre hiérarchique de nos classes devient alors celui exposé à la figure 6.9 .

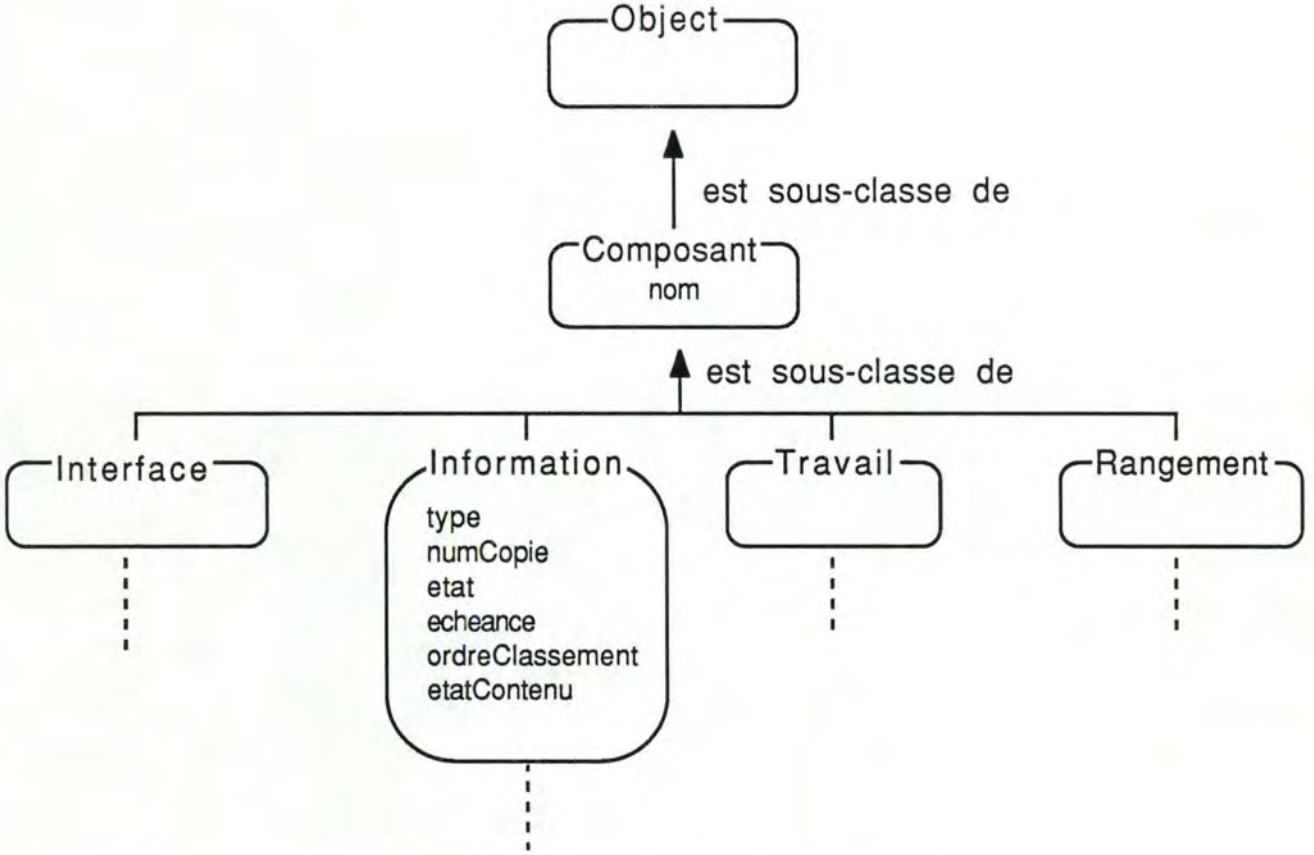


figure 6.9 : Hiérarchie des classes utilisées

6.3 La classe Bureau

6.3.1 La classe

Telle que nous l'avons exposée au point 6.2, notre hiérarchie de modèles est encore incomplète. En effet,

1) On peut envisager qu'un analyste désire analyser en parallèle plusieurs bureaux contenant des objets différents. Nous avons dès lors besoin d'objets permettant de mémoriser les informations relatives à un bureau particulier.

2) L'objectif de ce projet est de réaliser un outil permettant de réaliser des simulations graphiques de haut niveau de tâches de bureau, et ce pour un bureau particulier.

Or comme nous l'avons dit au début de ce chapitre, réaliser de telles applications nécessite l'utilisation de l'architecture M.V.C. Intuitivement, nous aurons donc besoin d'un modèle représentant le bureau et fournissant les données nécessaires aux simulations, d'une vue permettant de montrer la simulation et d'un contrôleur permettant à l'utilisateur de communiquer avec la vue.

Nous ajoutons donc à notre hiérarchie la classe Bureau qui décrit les structures de données permettant de conserver l'ensemble des objets informationnels,

d'interface, de rangement et de travail se trouvant dans un bureau donné, et donc associés à une instance de la nouvelle classe Bureau. Elle fournira également les messages nécessaires à la vue et au contrôleur afin d'obtenir les données à afficher.

Cette classe aura la classe Object pour super-classe, notre nouvelle classe n'ayant besoin que des variables et méthodes de base nécessaires à tous les modèles.

6.3.2 Les variables d'instance

Une instance de la classe Bureau posséde les variables d'instance suivantes:

* **nom**

  - cette variable contient le nom du bureau donné par l'analyste lors de sa création.
  - la valeur de cette variable est une instance de la classe Symbol (point 4.1.2.2).

* **information**

  - cette variable contient la liste des objets informationnels appartenant au bureau.
  - la valeur de cette variable est une instance de la classe Dictionary (point 4.1.2.2) organisée de la manière suivante:
  - les clés sont des instances de la classe Symbol (point 4.1.2.2) et désignent des noms de classe d'objets informationnels.
  - les valeurs sont des collections d'objets informationnels dont la classe est spécifiée par la clé.

* **rangement**

  - cette variable contient la liste de tous les objets de rangement appartenant au bureau.
  - la valeur de cette variable est une instance de la classe Dictionary (point 4.1.2.2). Ce dictionnaire est organisé de la même façon que celui contenu dans la variable information.

* **travail**

  - cette variable contient la liste de tous les objets de travail appartenant au bureau.
  - la valeur de cette variable est une instance de la classe Dictionary (point 4.1.2.2). Ce dictionnaire est organisé de la même façon que celui contenu dans la variable information.

* **interface**

  - cette variable contient la liste de tous les objets d'interface appartenant au bureau.

- la valeur de cette variable est une instance de la classe Dictionary (point 4.1.2.2). Ce dictionnaire est organisé de la même façon que celui contenu dans la variable information.

La hiérarchie complète des classes de modèle de notre programme apparait à la figure 6.10.

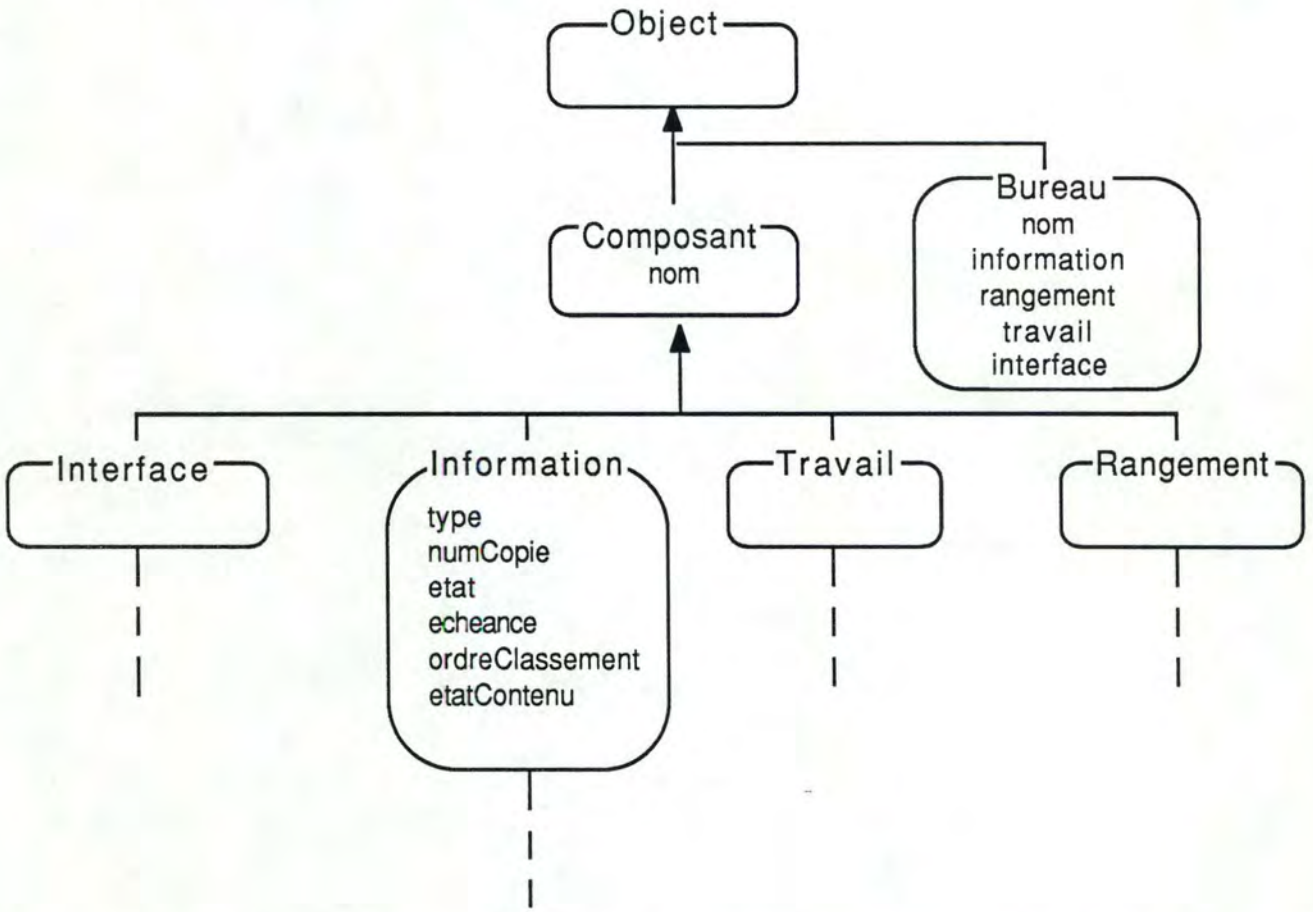


Figure 6.10: Hiérarchie des classes incluant la classe Bureau

6.4 Les identifiants des objets de bureau

6.4.1 Les objets informationnels

Pour un bureau particulier, l'identifiant de tout objet informationnel est composé de:

- sa classe
- son nom
- son numéro de copie
- son type

Le système assure l'unicité de cet identifiant pour chaque bureau.

6.4.2 Les objets de rangement

Pour un bureau particulier, l'identifiant de tout objet de rangement est composé de:

- sa classe
- son nom

Le système assure l'unicité de cet identifiant pour chaque bureau.

6.4.3 Les objets d'interface

Pour un bureau particulier, l'identifiant de tout objet d'interface est composé de:

- sa classe
- son nom

Le système assure l'unicité de cet identifiant pour chaque bureau.

6.4.4 Les objets de travail

Pour un bureau particulier, l'identifiant de tout objet de travail est composé de:

- sa classe
- son nom

Le système assure l'unicité de cet identifiant pour chaque bureau.

Il est à remarquer que ce sont également ces identifiants qui serviront de base pour la recherche d'un objet particulier dans un bureau donné.

6.5 La définition d'un environnement de bureau

Jusqu'à présent, nous avons défini des classes d'objets, ces objets étant les modèles de notre application. Nous avons associé des variables à chacune de ces classes afin de déterminer la partie statique des objets mais nous n'avons pas encore spécifié le moindre élément de leur protocole. C'est pourquoi dans les deux points qui vont suivre nous allons détailler les messages que doit définir l'analyste pour définir un environnement de bureau et décrire des tâches à simuler.

Avant de pouvoir simuler une tâche, l'analyste doit créer un bureau et son environnement. Pour cela, il exécute, de la manière décrite au point 3.4.5, un texte composé d'expressions Smalltalk. Ce texte doit contenir:

- la création d'un bureau
- la création des objets appartenant à ce bureau
- la création des associations entre ces objets

Pour rappel, les associations entre les objets sont de cinq types (point 2.4.3.2):

- un objet informationnel *est rangé dans* un objet de rangement
- un objet informationnel *est classé dans* un objet informationnel structurant
- un objet informationnel *est transmis* au moyen d'un objet d'interface
- un objet informationnel *est traité sur* un objet de travail
- un objet de rangement *est composé* d'un ou plusieurs objets de rangement

6.5.1 La création d'un bureau

La création d'un bureau signifie que l'on va créer une instance de la classe Bureau à l'aide d'un message. En Smalltalk, les messages de création d'instances sont déclarés dans les classes sous forme de messages de classe (point 3.4.2.2). Ceci implique que de tels messages doivent être envoyés à la classe dont on veut créer une instance. C'est ce principe qui a été également appliqué dans notre programme.

Le message de création d'une instance de la classe Bureau a donc été implémenté dans la classe Bureau. Il s'agit du message:

creer: unNom

où unNom est une instance de la classe Symbol (point 4.1.2.2) représentant le nom du bureau. L'exécution de la méthode déclenchée par l'envoi de ce message à la classe Bureau a pour effet de créer une instance de cette même classe, la variable *nom* de l'instance aura reçu la valeur unNom. Cette instance étant un objet, elle peut être affectée à une variable.

exemple:

```
b <- Bureau creer: #secretariat
```

L'exécution de cette expression affecte à b une instance de la classe Bureau dont le nom est #secretariat.

6.5.2 La création des objets d'un bureau

Une fois qu'il a créé une instance de la classe Bureau, l'analyste passe à la rédaction du texte de création des objets appartenant à ce bureau. L'appartenance d'un objet à un bureau laisse sous-entendre une association entre ces deux objets (figure 6.11).

Le schéma de la figure 6.11 spécifie d'une part qu'un objet de bureau appartient à un et un seul bureau et d'autre part qu'un bureau peut contenir un nombre quelconque d'objets.

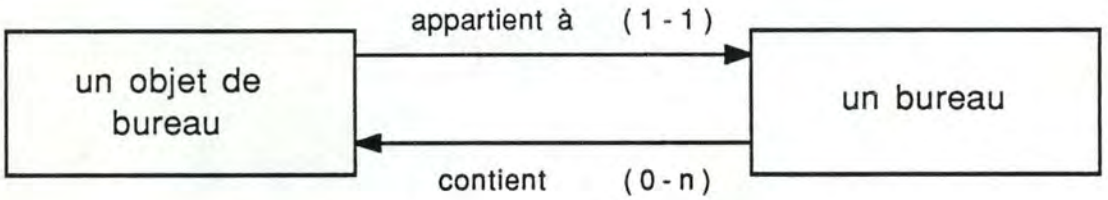


figure 6.11: L'association d'appartenance

6.5.2.1 L'association d'appartenance entre un objet et un bureau

Les messages de création d'objets, quelle que soit la classe des objets, contiennent un paramètre qui spécifie le bureau auquel l'objet appartient. Lorsqu'un objet est créé, il est ajouté au dictionnaire approprié (point 6.3.2) du bureau spécifié. Cela signifie qu'un objet informationnel sera inséré dans le dictionnaire *information* du bureau, un objet d'interface dans le dictionnaire *interface*, un objet de rangement dans le dictionnaire *rangement* et un objet de travail dans le dictionnaire *travail*. Le bureau possède alors une référence directe vers chacun de ses objets. Mais un objet a également besoin d'une référence pour connaître le bureau auquel il appartient. Pour ce faire, tout objet dispose d'une variable d'instance supplémentaire appelée *bureau* . Cette variable reçoit l'instance de la classe Bureau spécifiée en argument dans le message de création.

Un exemple est illustré à la figure 6.12 .

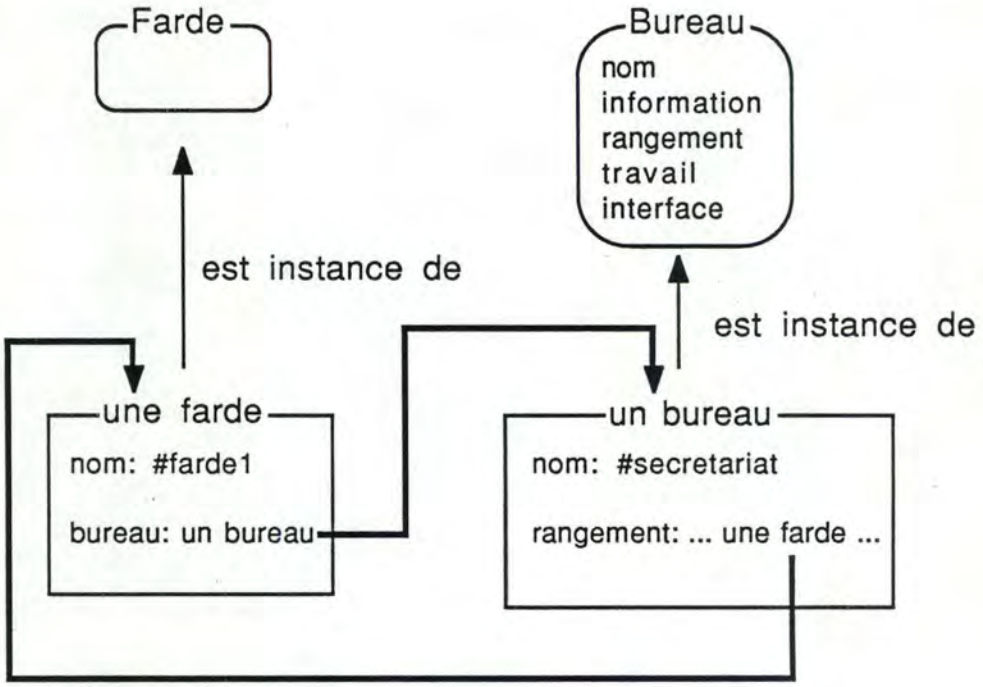


figure 6.12: Un exemple d'association d'appartenance

Puisque tout objet de bureau, quelle que soit sa classe, doit posséder une variable d'instance *bureau* , on peut appliquer les principes de généralisation et

d'abstraction.

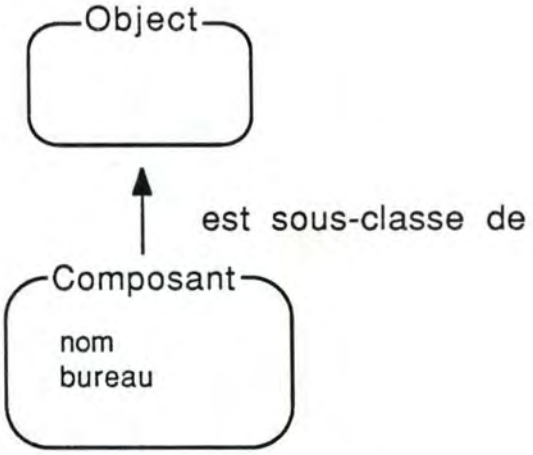


figure 6.13: La classe Composant

Ceci nous permet de remonter la déclaration de la variable *bureau* dans la classe Composant et toutes les autres classes en héritent alors automatiquement. La classe Composant devient alors telle qu'elle apparaît à la figure 6.13 .

6.5.2.2 La création des objets informationnels

Le message de création d'objet informationnel est envoyé à la classe dont on veut obtenir une instance. Il est identique quelle que soit la classe de l'objet; c'est pourquoi il a été implémenté dans la classe Information, toutes les sous-classes de cette classe en héritant automatiquement.

Le message de création d'un objet informationnel est de la forme suivante:

création: unNom **numCopie:** unNumCopie **type**: unType **état:** unEtat
échéance: uneEchéance **ordreClassement:** unOrdre **bureau:** unBureau

où

- unNom, unType, unEtat, unOrdre sont des instances de la classe Symbol (point 4.1.2.2).
- unNumCopie est une instance de la classe SmallInteger (point 4.1.2.1).
- uneEchéance est une instance de la classe String (point 4.1.2.2) représentant une date et est de la forme 'MM/JJ/AAAA'.
- unBureau est une instance de la classe Bureau.
- unOrdre doit être un des objets suivants:

  * #lifo
  * #fifo
  * #alphabétique
  * #alphabétiqueinverse
  * #aucun

Cette suite de caractères peut être indifféremment écrite en majuscules

et/ou en minuscules.

L'exécution de la méthode déclenchée par l'envoi de ce message à une sous-classe de la classe Information renvoie une instance de la classe réceptrice. Les variables d'instance de l'objet prennent les valeurs spécifiées en arguments et une association d'appartenance est créée entre l'objet et le bureau unBureau.

exemple:

```
f <- Formulaire création: #étudiant
                numCopie: 0
                type: #inscription
                état: nil
                échéance: nil
                ordreClassement: #aucun
                bureau: b.
```

L'exécution de cette expression affecte à f une instance de la classe Formulaire appartenant au bureau b et dont les valeurs des variables d'instance sont celles spécifiées en arguments.

6.5.2.3 La création des objets d'interface

Le message de création des objets d'interface est envoyé à la classe dont on veut obtenir une instance. Il est identique quelle que soit la classe de l'objet; c'est pourquoi il a été implémenté dans la classe Interface.

Le message de création d'un objet d'interface est de la forme suivante:

créerDans: unBureau

où unBureau est une instance de la classe Bureau.

L'exécution de la méthode déclenchée par l'envoi de ce message à une sous-classe de la classe Interface renvoie une instance de la classe réceptrice. La valeur de la variable *nom* de l'instance correspond au nom de la classe réceptrice. Cela signifie que l'analyste n'a pas le choix du nom d'un objet d'interface. Une association d'appartenance est créée entre l'objet d'interface et le bureau unBureau.

exemple:

```
tel <- Téléphone créerDans: b
```

L'exécution de cette expression affecte à la variable tel une instance de la classe Telephone appartenant au bureau b et dont la valeur de la variable d'instance nom est #Telephone.

6.5.2.4 La création des objets de travail

Tout ce qui a été écrit pour la création des objets d'interface est également

valable pour un objet de travail si ce n'est que la méthode correspondant au message créerDans a été implémentée dans la classe Travail.

Nous ne nous y attarderons donc pas.

6.5.2.5 La création des objets de rangement

Comme pour les autres objets de bureau, le message de création d'objets de rangement est envoyé à la classe dont on veut obtenir une instance. Le message est identique quelle que soit la classe de l'objet de rangement. C'est pourquoi il a été implémenté dans la classe Rangement. Toutes les sous-classes de cette classe en héritent donc automatiquement.

Le message de création d'objets de rangement est de la forme suivante:

créer: unNom **bureau:** unBureau

où

- unNom est une instance de la classe Symbol (point 4.1.2.2).
- unBureau est une instance de la classe Bureau.

exemple:

a <- Armoire créer: #armoire1 bureau: b

L'exécution de cette expression affecte à la variable a une instance de la classe Armoire appartenant au bureau b et dont la valeur de la variable d'instance nom est #armoire1.

6.5.2.6 Les contraintes

Lors de la création de l'environnement d'un bureau, certaines contraintes doivent être respectées. Le système se charge de les vérifier. Ces contraintes sont les suivantes:

- la présence d'une et une seule table de travail dans un bureau est obligatoire.
- une et une seule photocopieuse peut être présente dans un bureau
- cette dernière contrainte est également valable pour le téléphone,la poubelle, la boîte IN et la boîte OUT.

6.5.3 La création des associations entre les objets du bureau

Après avoir décrit les associations entre un objet et un bureau, nous allons à présent décrire les associations entre les objets d'un même bureau. Ces associations sont de cinq types possibles:

- l'association de composition entre deux objets de rangement
- l'association entre un objet informationnel et un objet d'interface

- l'association entre un objet informationnel et un objet de rangement
- l'association entre un objet informationnel et un objet de travail
- l'association de classement entre deux objets informationnels

Tous les messages de création d'association qui vont suivre font partie des protocoles de classe des différentes classes. Ils doivent donc être envoyés à la classe à laquelle ils appartiennent ou à une de ses sous-classes.

6.5.3.1 L'association de composition entre deux objets de rangement

Cette association permet de représenter le fait qu'un objet de rangement peut être composé d'autres objets de rangement. Par exemple, une armoire est composée de tiroirs.

La réalisation de cette association, comme celles qui vont suivre, adopte le même principe que celui appliqué pour l'association d'appartenance entre un objet et son bureau. En effet, les deux objets, que l'on appelera par la suite *composant* et *composé*, possèdent une référence directe vers l'objet qu'il compose et/ou vers les objets qui le composent. Ces références se font sous forme de variables d'instance.

Ainsi tout objet de rangement possède une variable *association* dans laquelle se trouve une instance de la classe Dictionary (point 4.1.2.1). Ce dictionnaire contient, rangés par classe d'objets, tous les objets de rangement qui le composent. Ce dictionnaire peut éventuellement être vide.

De la même façon, tout objet de rangement possède une variable *contenuDans*. Cette variable conserve une référence directe vers l'objet qu'il compose et est déclarée dans la classe Rangement.

exemple:

Supposons que l'on dispose d'une instance tiroir1 de la classe Tiroir et armoire1 de la classe Armoire. Si une relation de composition existe entre ces deux objets telle que le tiroir compose l'armoire, alors armoire1 possède l'objet tiroir1 dans son dictionnaire *association* et la valeur de la variable *contenuDans* de tiroir1 est l'objet armoire1.

La figure 6.14 illustre ces relations entre ces deux objets.

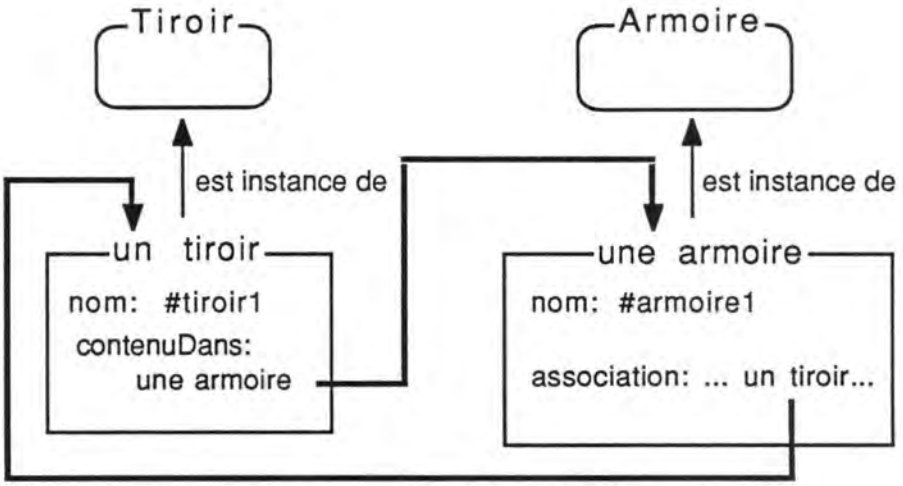


figure 6.14: Un exemple d'association de composition

Un certain nombre de contraintes [VPMS-87] doivent être vérifiées par le système:

- une armoire et une étagère ne peuvent être composant d'aucun autre objet de rangement
- un tiroir ne peut exister sans être composant d'une armoire.
- une farde ne peut exister sans être composant soit d'un tiroir, soit d'une boîte, soit d'une étagère.
- une armoire ne peut être composée que de tiroirs, leur nombre doit varier entre 3 et 6, pour des raisons de graphisme. Une étagère ne peut être composée que de boîtes ou de fardes. Un tiroir ou une boîte ne peut être composé que de fardes.

Le message de création d'association de composition fait partie du protocole de classe de la classe Rangement et toutes ses sous-classes en héritent. Il doit être envoyé à la classe de l'objet composant. Ce message est de la forme:

associationCompEntre: unNom1 **et:** uneClasseComposé **nomComposé:** unNom2 **bureau:** unBureau

où

- unNom1, unNom2 et uneClasseComposé sont des instances de la classe Symbol (point 4.1.2.2).
- unBureau est une instance de la classe Bureau.
- uneClasseComposé désigne le nom d'une sous-classe de Rangement.

La méthode déclenchée par l'envoi de ce message à une sous-classe de Rangement crée, dans le bureau unBureau, une association de composition entre une instance de la classe réceptrice, la variable *nom* de cette instance a la valeur unNom1, et une instance de la classe dont le nom est uneClasseCompose, et dont la variable d'instance *nom* a la valeur unNom2. Cette association ne sera réalisée qu'à la condition que ces deux objets de rangement existent dans le bureau unBureau.

Le premier objet spécifié est le composant et le second est le composé.

exemple:

Tiroir associationCompEntre: #tiroir1
et: #Armoire
nomCompose: #armoire1
bureau: b.

Cette expression a pour effet de créer, dans le bureau b, une association de composition entre une instance de la classe Tiroir et une instance de la classe Armoire à condition que ces objets existent dans le bureau spécifié. Les variables *nom* des instances de la classe Tiroir et Armoire ont respectivement les valeurs #tiroir1 et #armoire1. Le tiroir est le composant et l'armoire le composé.

6.5.3.2 L'association d'interface entre un objet informationnel et un objet d'interface

Grâce à cette association, on représente le fait qu'un objet informationnel est communiqué par un objet d'interface, de ou vers un bureau.

Pour réaliser cette association, tout objet informationnel dispose d'une variable d'instance *communiquePar*. La valeur de cette variable est l'objet d'interface par lequel est communiqué l'objet informationnel. Cette variable est déclarée dans la classe Information.

Quant à l'objet d'interface, il a à sa disposition un dictionnaire *association* dans lequel sont rangés, par classe, tous les objets informationnels qu'il communique.

De cette manière, les deux objets possèdent une référence directe vers l'objet auquel ils sont associés.

La réalisation de cette association impose un certain nombre de contraintes:

- seule une pile et une seule peut être associée à la boîte IN ou à la boîte OUT. Cette pile leur est toujours associée, même lorsque la boîte IN ou OUT est vide. La pile sera créée et l'association réalisée automatiquement par le système lors de la création de la boîte IN ou de la boîte OUT.
- un même objet informationnel ne peut participer que de manière exclusive aux types d'association d'interface, de classement et de rangement.
- un objet informationnel participe à au moins un type d'association d'interface, de traitement, de classement, de rangement.

Le message de création d'association d'interface est implémenté dans le protocole de classe de la classe Information et est donc hérité par toutes ses sous-classes. Il doit être envoyé à une des sous-classes d'InfoElementaire (point 6.1.1.1) ou d'InfoStructurant (point 6.1.1.1) et est de la forme:

associationIntEntre: unInfoNom **numCopie:** unNumCopie **type:** unType **et:**

unNomClasse **bureau:** unBureau

où

- unInfoNom, unType et unNomClasse sont des instances de la classe Symbol (point 4.1.2.2).
- unNumCopie est une instance de la classe SmallInteger (point 4.1.2.1).
- unBureau est une instance de la classe Bureau.
- unClasseNom désigne le nom d'une sous-classe de la classe Interface.

La méthode déclenchée par l'envoi de ce message à une sous-classe de InfoElementaire ou InfoStructurant crée, dans le bureau unBureau, une association d'interface entre une instance de la classe réceptrice-les variables nom, type et numCopie de cette instance prennent respectivement les valeurs unInfoNom, unType et unNumCopie-, et l'instance de la classe dont le nom est unNomClasse appartenant au bureau unBureau. Les deux objets spécifiés doivent appartenir au bureau unBureau.

exemple:

```
Dossier associationIntEntre: #dossier1
        numCopie: 0
        type: #inscription
        et: #BoiteOut
        bureau: b.
```

L'exécution de cette expression a pour effet de créer une association d'interface entre une instance de la classe Dossier et la boîte OUT du bureau b. La figure 6.15 décrit les relations existants entre les deux objets après l'exécution de l'expression.

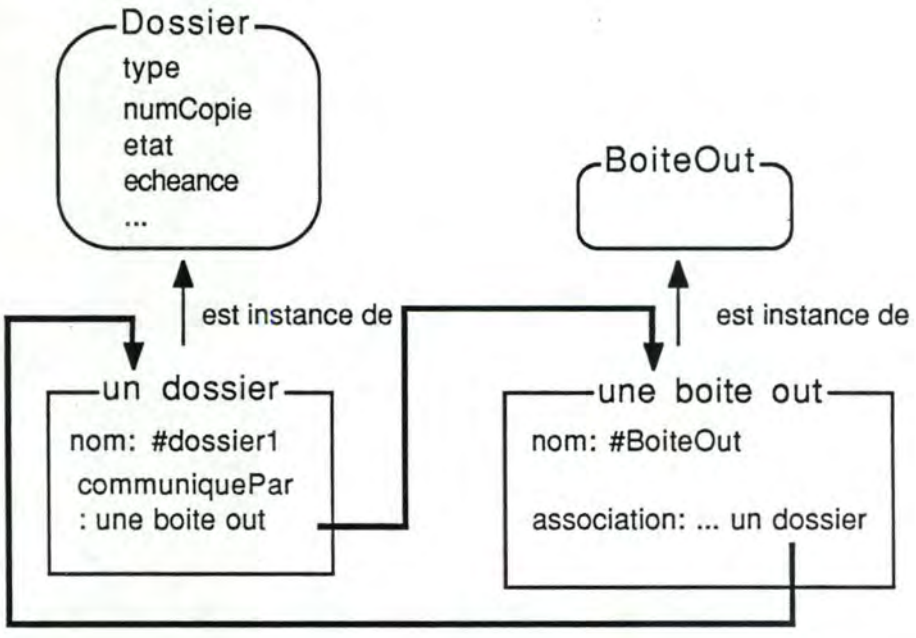


figure 6.15: Un exemple d'association d'interface

6.5.3.3 L'association de rangement entre un objet informationnel et un objet de rangement

L'association de rangement représente le rangement d'un objet informationnel élémentaire ou structurant dans un objet de rangement.

Le principe de réalisation de cette association est identique au principe de réalisation de l'association précédente. Un objet informationnel, qu'il soit élémentaire ou structurant, possède une variable d'instance *contenuDans* . La valeur de cette variable est l'objet de rangement dans lequel est rangé l'objet informationnel. Cette variable est déclarée dans la classe Information.

De son côté, l'objet de rangement possède un dictionnaire *association* dans lequel seront rangés, par classe, tous les objets informationnels qu'il contient.

Le protocole de création d'association de rangement est implémenté dans le protocole de classe de la classe Information et est donc hérité par toutes ses sous-classes. Il doit être envoyé à une des sous-classes de la classe InfoElementaire ou InfoStructurant et est de la forme:

associationRgtEntre: unNom1 **numCopie:** unNumCopie **type:** unType **et:** unNomClasse **nom:** unNom2 **bureau:** unBureau

où

- unNom1, unNom2, unType et unNomClasse sont des instances de la classe Symbol (point 4.1.2.2).
- unNumCopie est une instance de la classe SmallInteger (point 4.1.2.1).
- unBureau est une instance de la classe Bureau.
- unNomClasse désigne le nom d'une sous-classe de la classe Rangement.

La méthode déclenchée par l'envoi de ce message à une sous-classe de InfoElementaire ou InfoStructurant crée, dans le bureau unBureau, une association de rangement entre une instance de la classe réceptrice-les variables nom, type et numCopie de cette instance prennent respectivement les valeurs unNom1, unType et numCopie-, et une instance de la classe dont le nom est unNomClasse; la variable nom de cette deuxième instance prend la valeur unNom2. Ces deux objets doivent appartenir au bureau unBureau.

Les contraintes à respecter sont:

- aucun objet informationnel élémentaire ou structurant ne peut être rangé directement dans une armoire mais seulement dans les tiroirs de celle-ci.
- un même objet informationnel ne peut participer que de manière exclusive aux types d'association interface, classement et rangement.
- un objet informationnel participe à au moins un type d'association interface, traitement, classement, rangement.

exemple:

Formulaire associationRgtEntre: #étudiant
numCopie: 0
type: #inscription
et: #Farde
nom: #farde1
bureau: b.

Cette expression a pour effet de créer une association de rangement entre un formulaire et une farde appartenant tous les deux au bureau b. Les variables d'instance nom, numCopie et type du formulaire ont respectivement les valeurs #étudiant, 0 et #inscription. Quant à la variable d'instance nom de la farde, elle a la valeur #farde1 (figure 6.16).

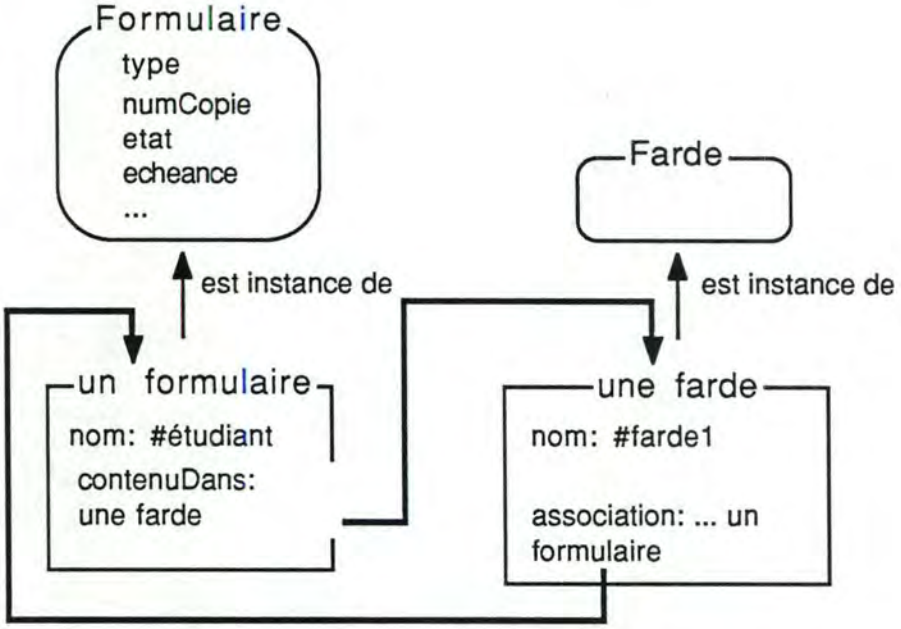


figure 6.16: Un exemple d'association de rangement

6.5.3.4 L'association de travail entre un objet informationnel et un objet de travail

Cette association représente le traitement ou la manipulation d'un objet informationnel sur un objet de travail.

Une association de ce type sera réalisée de la manière suivante: un objet informationnel a à sa disposition une variable d'instance *traiteSur* destinée à contenir l'objet de travail sur lequel est traité l'objet informationnel. Pour sa part, l'objet de travail possède un dictionnaire *association* où sont mémorisés, par classe, les objets informationnels qui sont traités sur l'objet de travail.

Les contraintes à respecter sont:

- un même objet informationnel ne peut participer que de manière exclusive aux types d'association interface, classement et rangement.
- un objet informationnel participe à au moins un type d'association interface, traitement, classement, rangement.

Le message de création de l'association de travail est implémenté dans le protocole de la classe Information et est donc hérité par toutes les sous-classes de cette classe. Il doit être envoyé à une sous-classe de InfoElementaire ou InfoStructurant et est de la forme:

associationTravailEntre: unInfoNom **numCopie:** unNumCopie **type:** unType **et:** unNomClasse **bureau:** unBureau

où

- unInfoNom, unType, unNomClasse sont des instances de la classe Symbol (point 4.1.2.2).
- unNumCopie est une instance de la classe SmallInteger (point 4.1.2.1).
- unBureau est une instance de la classe Bureau.
- unNomClasse désigne le nom d'une sous-classe de la classe Travail.

La méthode déclenchée par l'envoi de ce message à une sous-classe de InfoElementaire ou InfoStructurant crée une association de travail entre une instance de la classe réceptrice - les variables nom, type et numCopie de cette instance prennent respectivement les valeurs unNom1, unType et numCopie -, et une instance de la classe dont le nom est unClasseNom. Ces deux objets doivent appartenir au bureau unBureau.

exemple:

```
Dossier associationTravailEntre: #dossier
                numCopie: 0
                type: #étudiant
                et: #TableTravail
                bureau: b.
```

a pour effet de créer une association de travail entre le dossier dont l'identifiant est spécifié par les arguments #dossier, 0 et #étudiant et la table de travail du bureau b (figure 6.17).

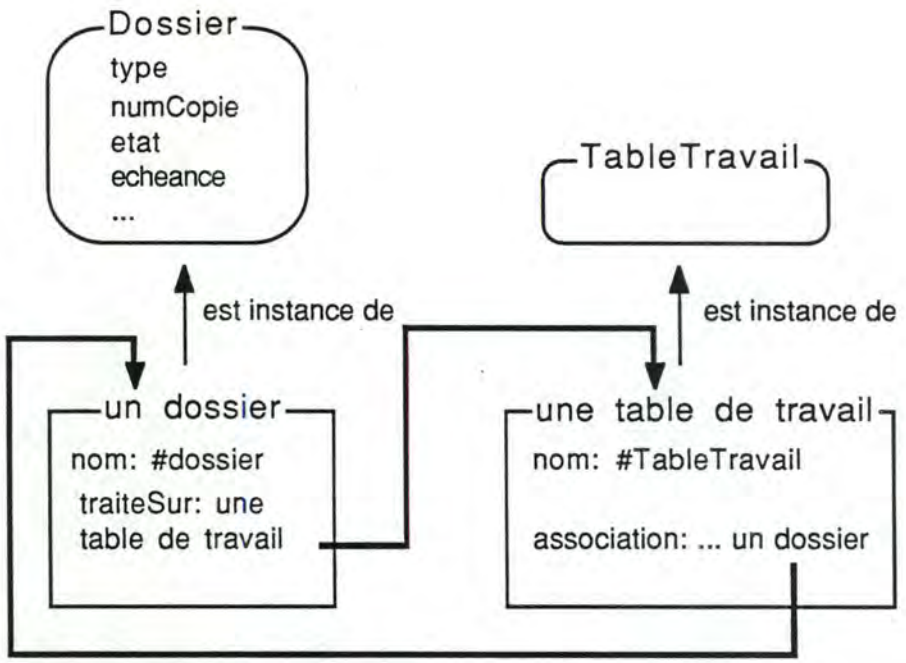


figure 6.17: Un exemple d'association de travail

6.5.3.5 L'association de classement entre deux objets informationnels

Cette association représente le classement d'un objet informationnel élémentaire ou structurant dans un objet informationnel structurant.

Le principe de réalisation d'une association de ce type est toujours le même avec cependant une petite variante. L'objet informationnel à classer a à sa disposition une variable d'instance *classeDans*, qui contient l'objet informationnel dans lequel il est classé. Dans le dictionnnaire *association* de l'objet informationnel de classement sont rangés, par classe, tous les objets informationnels qui le constituent.

Cependant, ceci ne permet pas de refléter correctement le classement des objets selon l'ordre spécifié. C'est pourquoi une variable *contenu* a été ajoutée à tout objet informationnel structurant. Cette variable contient l'ensemble des objets informationnels, toutes classes confondues, classés dans l'objet informationnel structurant mais selon l'ordre de classement spécifié à la création de ce dernier.

De plus afin de gérer correctement les ordres de classement FIFO et LIFO, tout objet informationnel faisant l'objet d'une association de classement reçoit un numéro d'entrée, ce numéro sera mémorisé dans une variable d'instance *numEntree* .

En définitive, tout objet informationnel possède donc les variables d'instance *association*, *classeDans* et *numEntree*, et tout objet informationnel structurant possède une variable d'instance *contenu* .

Suivant que l'on désire classer un objet informationnel élémentaire ou un objet informationnel structurant, on disposera de deux messages. Le premier a été implémenté dans le protocole de classe de la classe InfoElementaire, le second dans

le protocole de classe de la classe InfoStructurant. Cela signifie que le premier ne pourra être envoyé qu'à une sous-classe de InfoElementaire et le deuxième à une sous-classe de InfoStructurant. Ces deux messages sont:

- **associationClstEntre:** unNomOis **numCopie:** unNumCopie **oieType:** unOieType **et:** unNomOisClasse **oisNom:** unOisNom **oisType:** un Type **bureau:** unBureau

et

- **associationClstEntre:** unNom **oisType:** unOisType **et:** unNomClasse **nom:** unOisNom **type:** unType **bureau:** unBureau

où

- unNomOis, unOieType, unNomOisClasse, unType, unNom, unOisType, unNomClasse, unOisNom sont des instances de la classe Symbol (point 4.1.2.2).
- unNumCopie est une instance de la classe SmallInteger (point 4.1.2.1).
- unBureau est une instance de la classe Bureau.
- unNomOisClasse et unNomClasse désignent des noms de sous-classes de la classe InfoStructurant.

La méthode déclenchée par l'envoi de ce message à une des sous-classes de InfoElementaire, pour le premier message, ou d'InfoStructurant, pour le second message permet de créer une association de classement entre les deux objets informationnels désignés par les arguments. Ces deux objets doivent appartenir au bureau unBureau. Le premier objet désigné dans le message est l'objet à classer, le second est l'objet de classement.

exemple:

```
Formulaire associationClstEntre: #étudiant
                numCopie: 0
                oieType: #inscription
                et: #Fichier
                oisNom: #étudiant
                oisType: #inscription
                bureau: b.
```

L'exécution de cette expression permet de créer une association de classement entre un formulaire et un fichier dans le bureau b. Les variables d'instance nom, numCopie et type du formulaire ont les valeurs #étudiant, 0 et #inscription. Quant au fichier, les variables nom et type prennent les valeurs #étudiant et #inscription (figure 6.18).

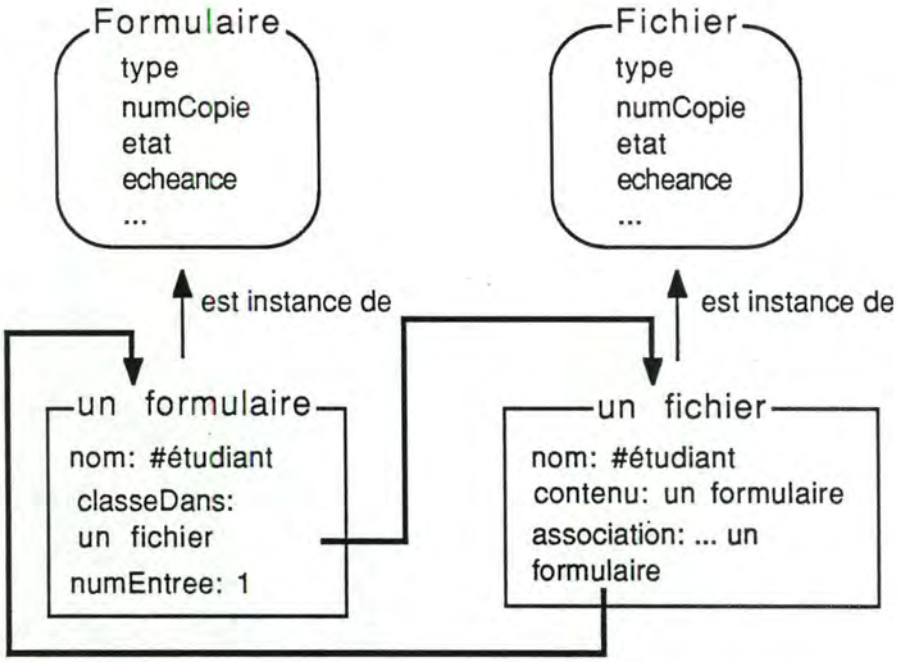


figure 6.18: Un exemple d'association de classement

Outre ces quatres types d'association, le système gère également, mais de manière automatique, un cinquième type d'association: une association de **copie**. Dans un bureau donné, un objet informationnel est la copie d'un autre s'il possède le même identifiant, tout en étant éventuellement caractérisé par des attributs différents.

6.5.3.6 Résumé des variables d'instance

Comme on a pu le voir dans les paragraphes précédents, tout objet de bureau, quelle que soit sa classe, possède une variable *association* . Une nouvelle fois, on peut introduire un niveau de généralisation supplémentaire. En effet, plutôt que d'associer une variable *association* à chacune des classes Information, Rangement, Travail et Interface, on peut déclarer cette variable dans la classe Composant puisque toutes ses sous-classes en hériteront automatiquement (cfr figure 6.10).

En fonction de ce qui a été écrit précédemment, nous pouvons dresser l'arbre hiérarchique des classes et de leurs variables d'instance associées (figure 6.19).

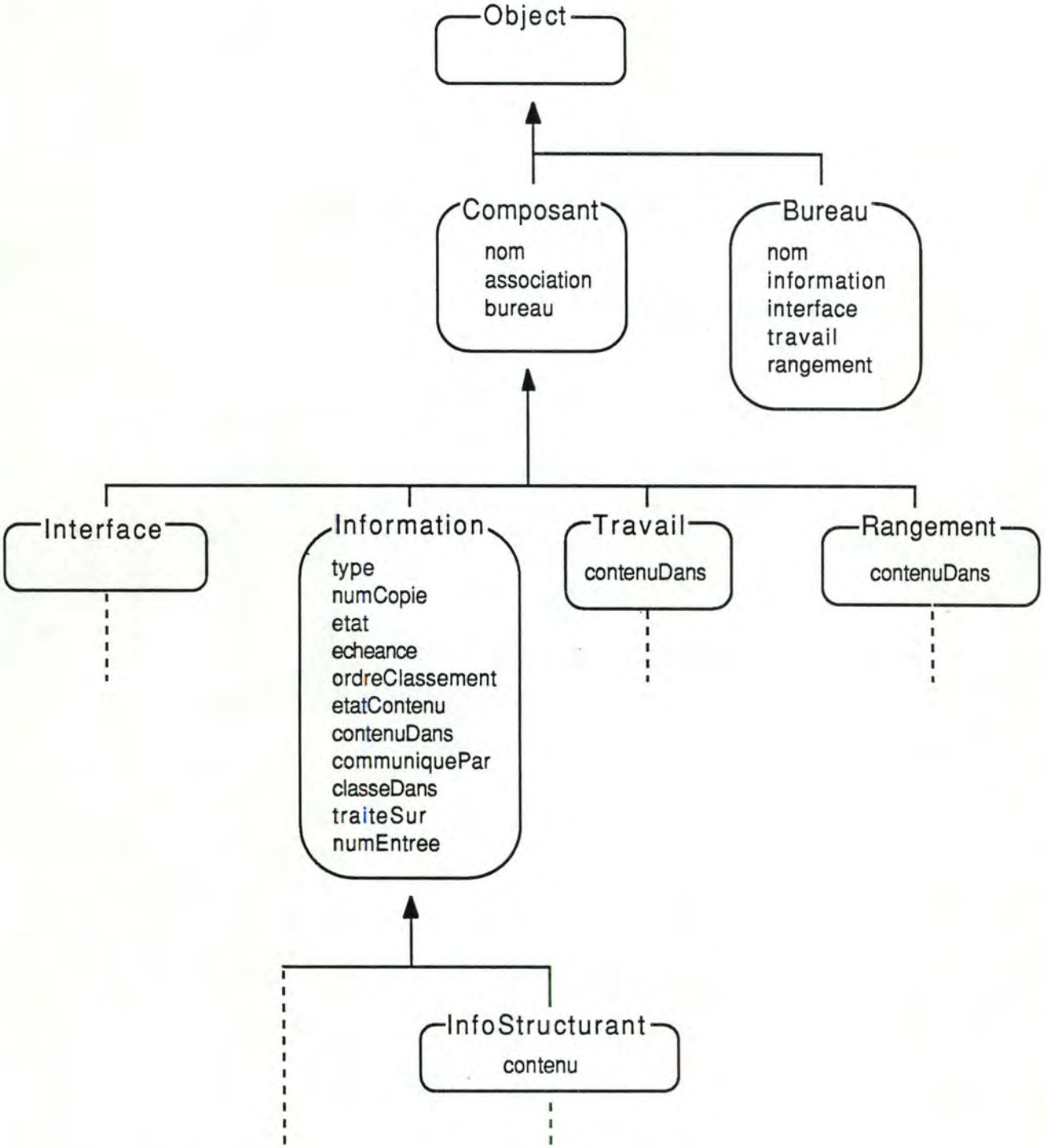


figure 6.19: Hiérarchie des classes de notre application et de leurs variables d'instance

6.6 La définition d'une tâche de bureau

Après avoir défini un environnement de bureau, l'analyste passe normalement à la phase de description des tâches à simuler. La description d'une tâche est contenue dans un texte composé d'expressions Smalltalk. La simulation de cette tâche est déclenchée par l'exécution de ce texte.

Comme nous l'avons vu au chapitre 2, il est possible de décomposer une tâche en opérations primitives selon des taxonomies. Comme ces opérations n'ont aucun lien entre elles, des structures d'enchaînement doivent être spécifiées. En Smalltalk, l'exécution des opérations correspondent à des envois de messages à des objets. Ce mécanisme est également appliqué pour les structures d'enchaînement (point 3.4.4).

Nous avons tenté de refléter la philosophie du langage le plus possible. C'est pourquoi, un problème est apparu lors de la rédaction des messages Smalltalk, à savoir la possibilité d'introduire des arguments facultatifs dans une phrase du langage (point 2.4.2). En effet, définir un sélecteur avec un nombre d'arguments variable est impossible en Smalltalk. Pour résoudre le problème et donc pour éviter à l'analyste de spécifier tous les arguments, nous avons associé différents messages à chaque opération primitive et structure d'enchaînement. Chacun des messages correspond alors à une combinaison possible d'arguments facultatifs.

Avant de passer à l'exposé des opérations primitives disponibles, nous allons examiner comment les concepts de variable et de paramètre du langage de description [VPMS-87] ont été introduits en Smalltalk (point 2.4.4.4).

6.6.1 Les concepts de variable et de paramètre

6.6.1.1 le concept de variable

Tout comme dans le langage de haut niveau, une variable permet de référencer l'identifiant d'un objet informationnel déterminé. Cet identifiant comporte la classe de l'objet, son nom, son numéro de copie et son type (point 6.1.1.3). L'utilisation de la variable permet à l'analyste de travailler sur des objets informationnels dont il ne connait pas l'identifiant. Tout comme dans le langage de spécification, nous verrons qu'il existe deux manières d'attribuer l'identifiant d'un objet informationnel à une variable: l'utilisation du composant de type boucle (point 2.4.4.6) et l'exécution de l'opération primitive *enlever*.

En Smalltalk, la variable sera perçue comme un objet. Ceci implique qu'on ne pourra accéder directement aux valeurs contenues dans une variable si ce n'est par l'utilisation de messages.

D'un point de vue implémentation, une variable est une instance de la classe OrderedCollection, c'est-à-dire une collection d'objets. Ces objets sont, dans l'ordre, un symbôle représentant le nom de la classe de l'objet, un symbôle représentant son nom, un entier représentant son numéro de copie et un symbôle représentant son type.

6.6.1.2 La concept de paramètre

Un paramètre désigne un emplacement de la mémoire destiné à recevoir une réponse de l'utilisateur. La valeur du paramètre est déterminée par les opérations de vérification ou de décision. Un paramètre, dans notre implémentation est considéré comme une simple variable Smalltalk. Elle peut donc être manipulée à l'aide des structures conditionnelles de contrôle classiques (point 3.4.4.2).

Exemple:

Soient les expressions suivantes:

```
paramètre1 <- Formulaire vérifier: #étudiant
                         numCopie: 3
                         type: #inscription
                         bureau: b.

(paramètre1 = #INCOMPLET)
        ifTrue: [bloc1]
        ifFalse: [bloc2].
```

La première expression de cet exemple permet de vérifier la complétude d'un formulaire et range le résultat de l'évaluation dans paramètre1. La seconde spécifie que si le formulaire est incomplet (paramètre1 = #incomplet) alors la suite d'expressions composant le bloc1 est exécutée. Si par contre, il est complet, la suite d'expressions du bloc2 est évaluée.

6.6.2 Les opérations primitives

Au point 2.3, nous avons dressé la liste des opérations applicables aux objets informationnels élémentaires ou structurants. Nous détaillons, dans ce paragraphe, les messages à utiliser pour déclencher l'exécution de chaque opération primitive.

Tous les messages décrits font partie du protocole de classe de la classe Information et sont donc hérités par toutes ses sous-classes, en particulier, par les classes Dossier, Fichier, Pile, Message, Formulaire et Document.

Pour chaque opération primitive, sont spécifiés:

* le ou les messages à utiliser pour déclencher son exécution,
* les contraintes syntaxiques portant sur les arguments,
* les préconditions d'application, c'est-à-dire les conditions que doivent respecter les arguments avant toute exécution de l'opération primitive pour que celle-ci s'exécute correctement,
* les postconditions d'application, c'est-à-dire les propriétés que les résultats doivent présenter en fin de toute exécution de l'opération primitive si celle-ci s'est exécutée correctement,
* un exemple d'application.

6.6.2.1 L'opération de **création** d'un objet informationnel

° Cette opération permet de créer un objet informationnel de type quelconque au sein d'une tâche. Son exécution est déclenchée par le message:

créer: unNom **numCopie:** unNumCopie **type:** unType **état:** unEtat
échéance: uneEchéance **ordreClassement:** unOrdre **bureau:** unBureau.

où

* unNom, unType, unEtat et unOrdre sont des instances de la classe Symbol (point 4.1.2.2).
* unNumCopie est une instance de la classe SmallInteger (point 4.1.2.1).
* unBureau est une instance de la classe Bureau.
* uneEchéance est une instance de la classe String (point 4.1.2.2) de la forme 'MM/JJ/AAAA'.
* si unOrdre = nil, le système remplacera cette valeur par #aucun. Sinon, unOrdre ne peut prendre (en majuscules et/ou en minuscules) que les valeurs #fifo, #lifo, #alphabétique, #alphabétiqueinverse.

° Précondition

- L'objet informationnel, spécifié par la classe réceptrice et les arguments, n'existe pas dans l'environnement du bureau unBureau.

° Postcondition

- L'objet informationnel spécifié est créé et est associé à la table de travail du bureau unBureau.

° Exemple

```
Dossier créer: #réponse
        numCopie: 0
        type: #typeRéponse
        état: #créé
        échéance: nil
        ordreClassement: #fifo
        bureau: b.
```

L'exécution de cette expression permet de créer, dans le bureau b, un dossier de nom #réponse, de numéro de copie 0, de type #typeRéponse, d'état #créé et d'ordre de classement #fifo. La valeur nil pour l'échéance signifie qu'aucune date d'échéance n'est associée au dossier créé.

6.6.2.2 L'opération de **réception d'un objet informationnel élémentaire** par la boîte IN

° Cette opération permet de recevoir un objet informationnel élémentaire par la boîte IN du bureau. Son exécution est déclenchée par le message:

RecevoirOie: unNom **numCopie:** unNumCopie **type:** unType **état:** unEtat **échéance:** uneEchéance **bureau:** unBureau.

où

* unNom, unType et unEtat sont des instances de la classe Symbol (point 4.1.2.2).
* unNumCopie est une instance de la classe SmallInteger (point 4.1.2.1).
* unBureau est une instance de la classe Bureau.

* uneEchéance est une instance de la classe String (point 4.1.2.2) de la forme 'MM/JJ/AAAA'.

° Préconditions

- La boîte IN existe dans l'environnement du bureau unBureau.
- L'objet informationnel élémentaire, spécifié par la classe réceptrice et les arguments, n'existe pas dans l'environnement du bureau unBureau.

° Postcondition

- L'objet informationnel élémentaire spécifié est créé et associé à la pile de la boîte IN du bureau unBureau.

° Exemple

Formulaire recevoirOie: #inscription
numCopie: 1
type: #typeInscription
état: #reçu
échéance: '12/31/1989'
bureau: b.

L'exécution de cette expression permet de recevoir dans le bureau b, un formulaire de nom #inscription, de numéro de copie 1, de type #typeInscription, d'état #reçu et de date d'échéance '12/31/1989'.

6.6.2.3 L'opération de **réception d'un objet informationnel structurant** par la boîte IN

° Cette opération permet de recevoir, en provenance de l'extérieur, un objet informationnel structurant, à l'exception d'une pile, par la boîte IN du bureau. Son exécution est déclenchée par le message:

RecevoirOis: unNom **numCopie:** unNumCopie **type:** unType **état:** unEtat **échéance:** uneEchéance **ordreClassement:** unOrdre **constituéDe:** unTabConstituants **bureau:** unBureau.

où

* unNom, unType, unEtat et unOrdre sont des instances de la classe Symbol (point 4.1.2.2).
* unNumCopie doit être une instance de la classe SmallInteger (point 4.1.2.1).
* unBureau est une instance de la classe Bureau.
* uneEchéance est une instance de la classe String (point 4.1.2.2) de la forme 'MM/JJ/AAAA'.
* si unOrdre = nil, le système remplacera cette valeur par #aucun. Sinon, unOrdre ne peut prendre (en majuscules et/ou en minuscules que les valeurs #fifo, #lifo, #alphabétique, #alphabétiqueinverse.

* unTabConstituants est un tableau contenant les informations permettant d'identifier et de qualifier chacun des composants de l'objet informationnel structurant reçu. Ce tableau est de la forme: #(objet1 ... objetn) où un objeti correspond à la suite d'informations suivantes:

    - un symbole spécifiant la classe du composant
    - un symbole spécifiant le nom du composant
    - un entier spécifiant le numéro de copie du composant
    - un symbole spécifiant le type du composant
    - un symbole spécifiant l'état du composant
    - un string pécifiant l'échéance du composant

° Préconditions

- La boîte IN existe dans l'environnement du bureau unBureau.
- L'objet informationnel structurant, spécifié par la classe réceptrice et les arguments, n'existe pas dans l'environnement du bureau unBureau.
- Aucun des composants spécifiés dans tabConstituants n'existe dans l'environnement du bureau unBureau.

° Postconditions

- L'objet informationnel structurant spécifié et ses constituants sont créés et associés à la pile de la boîte IN du bureau unBureau. Chacun des constituants est lié à l'objet structurant par une association de classement.
- Si l'ordre de classement spécifié est nil, le système remplace cette valeur par la valeur #aucun.
- Si l'ordre de classement spécifié pour l'objet structurant, est #alphabétique ou #alphabétiqueinverse, un tri sera réalisé sur les constituants. Si l'ordre est #lifo, #fifo, #aucun ou nil, l'ordre spécifié dans l'expression est sensé représenter cet ordre.

° Exemple

```
Dossier recevoirOis: #étudiant
        numCopie: 0
        type: #typeEtudiant
        état: #reçu

      échéance: '19/08/1988'
      ordreClassement: #aucun
      constituéDe:#(Document #introduction 0 #typeLettre #reçu '09/08/1987'
                    Formulaire #étudiant 3 #inscription #reçu '09/08/1987')
      bureau: b.
```

L'exécution de cette expression permet de recevoir dans le bureau b, un dossier de nom #étudiant, de numéro de copie 0, de type #typeEtudiant, d'état #reçu et de date d'échéance "19/08/1988' et composé d'un document et d'un formulaire.

6.6.2.4 La **réception** d'un message par le téléphone

° Cette opération permet de recevoir, en provenance de l'extérieur, un message par le téléphone. Selon le choix, on peut conserver ou non une trace écrite du message reçu. L'exécution de cette opération est déclenchée par le message:

RecevoirOralement: unNom **type:** unType **état:** unEtat **échéance:** uneEchéance **bureau:** unBureau.

où

* unNom, unType, unEtat sont des instances de la classe Symbol (point 4.1.2.2).
* unBureau est une instance de la classe Bureau.
* uneEchéance est une instance de la classe String (point 4.1.2.2) de la forme 'MM/JJ/AAAA'.

Si on ne désire pas garder de trace écrite du message, la valeur nil sera affectée aux arguments unNom, unType, unEtat et uneEchéance. Dans le cas contraire, les valeurs spécifiées constituent l'identifiant du message à créer dans l'environnement du bureau unBureau. Son numéro de copie vaut 0.

° Préconditions

- Le téléphone existe dans l'environnement du bureau unBureau.
- Si on désire conserver une trace écrite du message reçu, c'est-à-dire si les arguments unNom et unType ont reçu une valeur autre que nil, le message spécifié n'existe pas dans l'environnement du bureau unBureau.

° Postconditions

- Si on a désiré conserver une trace écrite du message reçu, le message spécifié est créé et est associé au téléphone du bureau unBureau par une association d'interface.
- Les valeurs différentes de nil des arguments unEtat et/ou uneEcheance sont affectées aux variables d'instance de l'objet informationnel, nouvellement créé.

° Exemple

```
Message recevoirOralement: #messageTéléphone
        type: #typeTéléphone
        état: #reçu
        échéance: nil
        bureau: b.
```

L'exécution de cette expression permet de recevoir un message par le téléphone du bureau b et d'en garder une trace sous forme d'un message de nom #messageTéléphone, de type #typeTéléphone et d'état #reçu.

6.6.2.5 La **communication** d'un objet informationnel par la boîte OUT

° Cette opération permet de communiquer, à l'extérieur et par la boîte OUT, un objet informationnel à l'exception de la pile. Pour déclencher l'exécution de cette opération, quatre messages sont disponibles selon que l'on désire utiliser une variable et/ou spécifier les arguments état et échéance. Ils ont les formes suivantes:

a) **communiquer:** uneVariable **bureau:** unBureau.
b) **communiquer:** uneVariable **état:** unEtat **échéance:** uneEchéance **bureau:** unBureau.
c) **communiquer:** unNom **numCopie:** unNumCopie **type:** unType **bureau:** unBureau.
d) **communiquer:** unNom **numCopie:** unNumCopie **type:** unType **état:** unEtat **échéance:** uneEchéance **bureau:** unBureau.

où

* unNom, unType, unEtat sont des instances de la classe Symbol (point 4.1.2.2).
* unNumCopie est une instance de la classe SmallInteger (point 4.1.2.1).
* unBureau est une instance de la classe Bureau.
* uneEchéance est une instance de la classe String (point 4.1.2.2) de la forme 'MM/JJ/AAAA'.
* uneVariable est une variable (point 6.6.1.1) qui a reçu une valeur grâce à une opération enlever ou à une boucle.

° Préconditions

- La boîte OUT existe dans l'environnement du bureau unBureau et une pile lui est associée.
- L'objet informationnel spécifié par la variable ou par les arguments, existe dans l'environnement du bureau unBureau.

° Postconditions

- L'objet informationnel est associé à la pile de la boîte OUT du bureau unBureau grâce à une association de classement.
- Si l'objet informationnel est du type structurant, et que certains de ses constituants sont sur un objet de travail, ces objets ne se trouvent pas dans l'objet d'interface.
- Les valeurs différentes de nil des arguments unEtat et/ouuneEcheance sont affectées aux variables d'instance de l'objet informationnel communiqué.

° Exemple

```
Dossier communiquer: #réponse
                numCopie: 0
                type: #typeRéponse
```

état: #envoyé
échéance: '09/05/1987'
bureau: b.

L'exécution de cette expression permet de communiquer, par la boîte OUT du bureau b un dossier de nom #réponse, de numéro de copie 0, de type #typeRéponse, d'état #envoyé et de date d'échéance '09/05/1987'.

6.6.2.6 La **communication** d'un message par le téléphone

° Cette opération permet d'envoyer, à l'extérieur, un message par le téléphone. Dans ce cas, il est possible de spécifier l'objet informationnel qui est à l'origine du message à communiquer. Dans ce message, une variable peut être utilisée. Si c'est le cas, elle identifie l'objet informationnel qui est à l'origine du message à communiquer. Pour déclencher l'exécution de cette opération, cinq messages sont disponibles:

a) **communiquerOralementDans:** unBureau.
b) **communiquerOralement:** uneVariable **bureau:** unBureau.
c) **communiquerOralement:** uneVariable **état:** unEtat **échéance:** uneEchéance **bureau:** unBureau.
d) **communiquerOralement:** unNom **numCopie:** unNumCopie **type:** unType **bureau:** unBureau.
e) **communiquerOralement:** unNom **numCopie:** unNumCopie **type:** unType **état:** unEtat **échéance:** uneEchéance **bureau:** unBureau.

où

* unNom, unType, unEtat sont des instances de la classe Symbol (point 4.1.2.2).
* unNumCopie est une instance de la classe SmallInteger (point 4.1.2.1).
* unBureau est une instance de la classe Bureau.
* uneEchéance est une instance de la classe String (point 4.1.2.2) de la forme 'MM/JJ/AAAA'.
* uneVariable est une variable (point 6.6.1.1) qui a reçu une valeur grâce à une opération enlever ou grâce à une boucle.

° Préconditions

- Le téléphone existe dans l'environnement du bureau unBureau.
- Si on a spécifié l'objet informationnel à l'origine du message, celui-ci existe dans l'environnement du bureau unBureau. En outre, cet objet ne peut être une pile.

° Postconditions

- Si on a spécifié l'objet informationnel à l'origine du message, l'objet informationnel est associé au téléphone du bureau par une association d'interface.

- Les valeurs différentes de nil des arguments unEtat et/ou uneEcheance sont affectées aux variables d'instance de l'objet informationnel.

° Exemple

Message communiquerOralement: #inscriptions
numCopie: 0
type: #typeInscriptions
état: #envoyé
échéance: nil
bureau: b.

L'exécution de cette expression permet de communiquer un message de nom #inscriptions, de numéro de copie 0, de type #typeInscriptions et d'état #envoyé par le téléphone du bureau b.

6.6.2.7 L'opération de **destruction** d'un objet informationnel

° Cette opération permet de détruire un objet informationnel en l'associant à la poubelle. Selon que l'on utilise une variable et/ou que l'on spécifie l'état et l'échéance de l'objet, on dispose de quatre messages différents pour déclencher l'exécution de cette opération:

a) **détruire:** uneVariable **bureau:** unBureau.
b) **détruire:** uneVariable **état:** unEtat **échéance:** uneEchéance **bureau:** unBureau.
c) **détruire:** unNom **numCopie:** unNumCopie **type:** unType **bureau:** unBureau.
d) **détruire:** unNom **numCopie:** unNumCopie **type:** unType **état:** unEtat **échéance:** uneEchéance **bureau:** unBureau.

où

* unNom, unType, unEtat sont des instances de la classe Symbol (point 4.1.2.2).
* unNumCopie est une instance de la classe SmallInteger (point 4.1.2.1).
* unBureau est une instance de la classe Bureau.
* uneEchéance est une instance de la classe String (point 4.1.2.2) de la forme 'MM/JJ/AAAA'.
* uneVariable est une variable (point 6.6.1.1) qui a reçu une valeur grâce à une opération enlever ou grâce à une boucle.

° Préconditions

- L'objet informationnel spécifié existe dans l'environnement du bureau unBureau.
- Une poubelle existe dans l'environnement du bureau unBureau.

° Postconditions

- L'objet informationnel spécifié est associé à la poubelle du bureau unBureau par une association d'interface.
- Les valeurs différentes de nil des arguments unEtat et/ou uneEcheance sont affectées aux variables d'instance de l'objet informationnel.
- L'objet informationnel spécifié n'est plus accessible.
- Si l'objet à détruire est un objet informationnel structurant et si certains de ses constituants sont associés à un objet de travail, ceux-ci ne sont pas détruits.

° Exemple

Information détruire: uneVariable
bureau: b.

L'exécution de cette expression permet de détruire, dans le bureau b, l'objet informationnel identifié par la variable uneVariable.

6.6.2.8 L'archivage d'un objet informationnel

° Cette opération permet d'archiver un objet informationnel quelconque, à l'exception d'un message et d'une pile, dans une boîte à archive. Selon que l'on utilise une variable et/ou que l'on spécifie l'état et l'échéance de l'objet , on dispose de quatre messages différents pour déclencher l'exécution de cette opération:

a) **archiver:** uneVariable **nomBoîte:** unNomBoîte **bureau:** unBureau.
b) **archiver:** uneVariable **état:** unEtat **échéance:** uneEchéance **nomBoîte:** unNomBoîte **bureau:** unBureau.
c) **archiver:** unNom **numCopie:** unNumCopie **type:** unType **nomBoîte:** unNomBoîte **bureau:** unBureau.
d) **archiver:** unNom **numCopie:** unNumCopie **type:** unType **état:** unEtat **échéance:** uneEchéance **nomBoîte:** unNomBoîte **bureau:** unBureau.

où

* unNom, unType, unEtat et unNumCopie sont des instances de la classe Symbol (point 4.1.2.2).
* unNumCopie est une instance de la classe SmallInteger (point 4.1.2.1).
* unBureau est une instance de la classe Bureau.
* uneEchéance est une instance de la classe String (point 4.1.2.2) de la forme 'MM/JJ/AAAA'.
* uneVariable est une variable (point 6.6.1.1) qui a reçu une valeur grâce à une opération enlever ou grâce à une boucle.

° Préconditions

- L'objet informationnel spécifié existe dans l'environnement du bureau unBureau.
- La boîte archive dont le nom est unNomBoîte existe dans l'environnement du bureau unBureau.

° Postconditions

- L'objet informationnel spécifié est associé à la boîte archive du bureau unBureau par une association de rangement.
- Les valeurs différentes de nil des arguments unEtat et/ou uneEcheance sont affectées aux variables d'instance de l'objet informationnel.
- Si l'objet à archiver est un objet informationnel structurant et si certains de ses constituants sont associés à un objet de travail, ceux-ci ne sont pas archivés.

° Exemple

Dossier archiver: #étudiant
numCopie: 0
type: #typeEtudiant
état: #archive
échéance: nil
nomBoîte: #archive
bureau: b.

L'exécution de cette expression permet d'archiver, dans le bureau b, le dossier spécifié par les arguments dans la boîte archive de nom #archive.

6.6.2.9 Le **classement** d'un objet informationnel dans un objet informationnel structurant

° Cette opération permet de classer un objet informationnel élémentaire ou structurant quelconque, à l'exception d'une pile, dans un objet informationnel structurant. pour déclencher l'exécution de cette opération, deux messages sont disponibles selon que l'on désire spécifier l'objet à classer explicitement ou par l'intermédiaire d'une variable:

a) **classer:** uneVariable **dans:** unOisClasseNom **oisNom:** unNom2 **oisType:** unOisType **bureau:** unBureau.

b) **classer:** unNom1 **numCopie:** unNumCopie type: unType **dans:** unOisClasseNom **oisNom:** unNom2 **oisType:** unOisType **bureau:** unBureau.

où

* unNom1, unType1, unOisClasseNom, unNom2, unOisType sont des

instances de la classe Symbol (point 4.1.2.2).
* unOisClasseNom désigne le nom d'une sous-classe de InfoStructurant.
* unBureau est une instance de la classe Bureau.
* unNumCopie est une instance de la classe SmallInteger (point 4.1.2.1).
* uneVariable est une variable (point 6.6.1.1) qui a reçu une valeur grâce à une opération enlever ou grâce à une boucle.

° Préconditions

- Les deux objets informationnels spécifiés existent dans l'environnement du bureau unBureau.
- Une pile ne peut être classée dans un autre objet informationel..

° Postconditions

- L'objet informationnel spécifié est classé dans l'objet informationnel structurant.
- Si l'objet à classer est un objet informationnel structurant et si certains de ses constituants sont associés à un objet de travail, ceux-ci ne sont pas classés.

° Exemple

Formulaire classer: #étudiant
numCopie: 3
type: #inscription
dans: #Fichier
oisNom: #inscriptions
oisType: nil
bureau: b.

L'exécution de cette expression permet de classer, dans le bureau b, le formulaire de nom #étudiant, de numéro de copie 3 et de type #inscription dans le fichier de nom #inscriptions.

6.6.2.10 Le **rangement** d'un objet informationnel dans un objet de rangement

° Cette opération permet de ranger un objet informationnel quelconque, dans un objet de rangement. Pour déclencher l'exécution de cette opération, deux messages sont disponibles selon que l'on spécifie l'objet à ranger explicitement ou par l'intermédiaire d'une variable.

a) **ranger:** uneVariable **dans:** unClasseNom **nom:** unRgtNom **bureau:** unBureau.

b) **ranger:** unNom **numCopie:** unNumCopie **type:** unType **dans:** unClasseNom **nom:** unRgtNom **bureau:** unBureau.

où

* unNom, unType, unClasseNom, unRgtNom sont des instances de la classe Symbol (point 4.1.2.2).
* unClasseNom désigne le nom d'une sous-classe de Rangement.
* unBureau est une instance de la classe Bureau.
* unNumCopie est une instance de la classe SmallInteger (point 4.1.2.1).
* uneVariable est une variable (point 6.6.1.1) qui a reçu une valeur grâce à une opération enlever ou à une boucle.

° Préconditions

- L'objet informationnel existe dans l'environnement du bureau unBureau.
- L'objet de rangement existe dans l'environnement du bureau unBureau.

° Postconditions

- L'objet informationnel spécifié est rangé dans l'objet de rangement.
- Si l'objet à ranger est un objet informationnel structurant et si certains de ses constituants sont associés à un objet de travail, ceux-ci ne sont pas rangés dans l'objet de rangement.

° Exemple

Formulaire ranger: #étudiant
numCopie: 0
type: #inscription
dans: #Farde
nom: #farde1
bureau: b.

L'exécution de cette expression permet de ranger, dans le bureau b, le formulaire de nom #étudiant, de numéro de copie 0 et de type #inscription dans la farde de nom #farde1.

6.6.2.11 Le **replacement** d'un objet informationnel

° Cette opération permet de remettre à leur place (lieu de classement, lieu de rangement ou lieu d'interface), les objets informationnels qui se trouvent sur la table de travail ou sur la photocopieuse. Il faut spécifier soit l'objet à replacer explicitement ou par l'intermédiaire d'une variable, soit une classe d'objets i nformationnels, soit tous les objets informationnels du bureau. Pour déclencher l'exécution de cette opération, divers messages sont disponibles:

1) Le replacement d'un objet informationnel spécifié

a) **replacer:** uneVariable **bureau:** unBureau.

b) **replacer:** unNom **numCopie:** unNumCopie **type:** unType **bureau:** unBureau.

2) Le replacement de plusieurs objets informationnels selon un critère précis

replacerTous: unType **bureau:** unBureau.

où

* uneVariable est une variable au sens où nous l'avons défini au point 6.6.1.1 et qui a reçu une valeur grâce à une opération enlever ou grâce à une boucle.
* unNom, unType sont des instances de la classe Symbol (point 4.1.2.2).
* unBureau est une instance de la classe Bureau.
* unNumCopie est une instance de la classe SmallInteger (point 4.1.2.1).

° Précondition

- Dans le cas des deux premiers messages, l'objet informationnel existe dans l'environnement du bureau unBureau et est associé soit à la table de travail, soit à la photocopieuse.

° Postcondition

- Les objets informationnels répondant au critère de sélection ont été replacés dans leur lieu de rangement, d'interface ou de classement.

° Exemples:

```
Formulaire replacer: #étudiant
                numCopie: 3
                type: #inscription
                bureau: b.
```

L'exécution de cette expression permet de replacer le formulaire de nom #étudiant, de numéro de copie 3 et de type #inscription dans le bureau b.

```
Formulaire replacerTous: #inscription
                bureau: b.
```

L'exécution de cette expression permet de replacer tous les formulaires de type #inscription dans le bureau b.

4.6.2.12 La **consultation** d'un objet informationnel

° Cette opération permet de consulter le contenu d'un objet informationnel élémentaire. Pour déclencher l'exécution de cette opération, quatre messages sont disponibles selon que l'on désire utiliser une variable et/ou spécifier les arguments état et échéance. Ils ont la forme suivante:

a) **consulter:** uneVariable **bureau:** unBureau.

b) **consulter:** uneVariable **etat:** unEtat **échéance:** uneEchéance **bureau:** unBureau.

c) **consulter:** unNom **numCopie:** unNumCopie **type:** unType **bureau:** unBureau.

d) **consulter:** unNom **numCopie:** unNumCopie **type:** unType **etat:** unEtat **échéance:** uneEchéance **bureau:** unBureau.

où

* unNom, unType sont des instances de la classe Symbol (point 4.1.2.2).
* unBureau est une instance de la classe Bureau.
* unNumCopie est une instance de la classe SmallInteger (point 4.1.2.1).
* uneVariable est une variable (point 6.6.1.1) qui a reçu une valeur grâce à une opération enlever ou à une boucle.

° Précondition

- L'objet informationnel spécifié existe dans l'environnement du bureau unBureau et est élémentaire.

° Postconditions

- L'objet informationnel spécifié a été consulté et est associé a la table de travail.
- Les valeurs, différentes de nil, des arguments unEtat et/ou uneEcheance sont affectées aux variables d'instance de l'objet informationnel.

° Exemple

```
Document consulter: #inscription
                    numCopie: 1
                    type: #typeInscription
                    bureau: b.
```

L'exécution de cette expression permet de consulter le document de nom #inscription, de numéro de copie 1 et de type #typeInscription dans le bureau b.

6.6.2.13 La **modification** d'un objet informationnel

° Cette opération permet de modifier le contenu d'un document à partir d'un message brouillon. Selon les paramètres que l'on veut préciser, différents messages sont disponibles pour déclencher l'exécution de cette opération:

a) **modifier:** uneVariable **auMoyenDe:** unMsgeNom **numCopieMsge:** unNumCopie **typeMsge:** unTypeMsge **bureau:** unBureau.

b) **modifier:** uneVariable **état:** unEtat **échéance:** uneEchéance **auMoyenDe:** unMsgeNom **numCopieMsge:** unNumCopie **typeMsge:** unTypeMsge **bureau:** unBureau.

d) **modifier:** unNom **numCopie:** unNumCopie **type:** unType **état:** unEtat **échéance:** uneEchéance **auMoyenDe:** unMsgeNom **numCopieMsge:** unNumCopie **typeMsge:** unTypeMsge **bureau:** unBureau.

où

* unNom, unType, unEtat, unMsgeNom, unTypeMsge sont des instances de la classe Symbol (point 4.1.2.2).
* unBureau est une instance de la classe Bureau.
* unNumCopie est une instance de la classe SmallInteger (point 4.1.2.1).
* la spécification du message est facultative. Si on ne ne veut pas le spécifier, il suffit d'affecter la valeur nil aux arguments unMsgeNom, unNumCopie et unTypeMsge
* uneVariable est une variable (point 6.6.1.1) qui a reçu une valeur grâce à une opération enlever ou grâce à une boucle.

° Préconditions

- Le document, spécifié par la variable ou par la classe réceptrice et les arguments, existe dans l'environnement du bureau unBureau.
- Le message brouillon spécifié existe dans l'environnement du bureau unBureau.

° Postconditions

- Le document spécifié a été modifié et est associé à la table de travail.
- Le message brouillon éventuel est associé à la table de travail.
- Les valeurs, différentes de nil, des arguments unEtat et/ou uneEcheance sont affectées aux variables d'instance de l'objet informationnel.

° Exemple

```
Information modifier: variable
                    état: #modifié
                    échéance: '08/31/1988'
                    auMoyenDe: #brouillon
                    numCopieMsge: 0
                    typeMsge: #typeBrouillon
                    bureau: b.
```

L'exécution de cette expression permet de modifier le document identifié par la variable variable, au moyen d'un message de nom #brouillon, de numéro de copie 0 et de type #typeBrouillon.

6.6.2.14 L'opération **compléter**

° Cette opération permet de compléter un formulaire en remplissant ses champs. Selon que l'on spécifie le formulaire à l'aide d'une variable et/ou des arguments état et échéance ou pas, différents messages sont disponibles

pour déclencher l'exécution de cette opération:

a) **compléter:** uneVariable **bureau:** unBureau.

b) **compléter:** uneVariable **état:** unEtat **échéance:** uneEchéance **bureau:** unBureau.

c) **compléter:** unNom **numCopie:** unNumCopie **type:** unType **bureau:** unBureau.

d) **compléter:** unNom **numCopie:** unNumCopie **type:** unType **état:** unEtat **échéance:** uneEchéance **bureau:** unBureau.

où

* unNom, unType, unEtat sont des instances de la classe Symbol (point 4.1.2.2).
* unBureau est une instance de la classe Bureau.
* unNumCopie est une instance de la classe SmallInteger (point 4.1.2.1).
* uneVariable est une variable (point 6.6.1.1) qui a reçu une valeur grâce à l'exécution d' une opération enlever ou grâce à l'utilisation du composant de type boucle.

° Précondition

- L'objet informationnel spécifié par la variable ou par la classe réceptrice et les arguments, existe dans l'environnement du bureau unBureau et ne peut être qu'un formulaire.

° Postconditions

- Le formulaire spécifié a été complété et est associé à la table de travail.
- Les valeurs, différentes de nil, des arguments unEtat et/ou uneEcheance sont affectées aux variables d'instance de l'objet informationnel.

° Exemple

```
Formulaire compléter: #inscription
                numCopie: 0
                type: #typeInscription
                état: #complété
                échéance: '01/29/1989'
                bureau: b.
```

L'exécution de cette expression permet de compléter le formulaire de nom #inscription, de numéro de copie 0 et de type #typeInscription dans le bureau b.

6.6.2.15 L'opération **décider**

° Cette opération permet de prendre une décision de routine sur base du contenu d'un formulaire, d'un document ou d'un dossier. Pour déclencher l'exécution de cette opération, différents messages sont disponibles selon que l'on spécifie l'objet à ranger explicitement ou par l'intermédiaire d'une variable.

a) **déciderAPartirDe:** uneVariable **bureau:** unBureau.

b) **déciderAPartirDe:** unNom **numCopie:** unNumCopie **type:** unType **bureau:** unBureau.

où

* unNom, unType sont des instances de la classe Symbol (point 4.1.2.2).
* unNumCopie est une instance de la classe SmallInteger (point 4.1.2.1).
* unBureau est une instance de la classe Bureau.
* uneVariable est une variable (point 6.7.1.1) et qui a reçu une valeur grâce à l'exécution d' une opération enlever ou grâce à l'utilisation du composant de type boucle.

° Précondition

- L'objet informationnel spécifié existe dans l'environnement du bureau unBureau et est un formulaire, un document ou un dossier.

° Postconditions

- L'objet informationnel spécifié est associé à la table de travail.
- La décision a été prise et peut donc être affectée à une variable Smalltalk. Cette opération met en oeuvre la notion de paramètre définie au point 6.7.1.2. La réponse renvoyée est une des valeurs suivantes:

#ACCEPTABLE
#NONACCEPTABLE.

° Exemple

```
paramètre <- Dossier déciderAPartirDe: #étudiant
                     numCopie: 0
                     type: #typeEtudiant
                     bureau: b.
```

L'exécution de cette expression permet de prendre une décision sur base du dossier nom #étudiant, de numéro de copie 0 et de type #typeEtudiant. Le résultat de la décision est affecté à la variable paramètre.

6.6.2.16 L'opération **feuilleter**

° Cette opération permet de feuilleter un objet informationnel structurant et donc de connaître chacun des objets informationnels élémentaires et structurants qui en sont constituants. Si certains des constituants de l' objet informationnel structurant spécifié sont traités sur un objet de travail et donc, ne sont pas physiquement dans l'objet informationnel structurant, ils ne sont pas montrés lors de cette opération.L'analyste a la possibilité de spécifier un ordre de consultation. Cela n'influence en rien l'ordre de classement initial qui est attaché à l'objet informationnel structurant. Si aucun ordre de consultation n'est spécifié, alors l'ordre de classement de l'objet informationnel est utilisé.Selon que l'on spécifie l'objet à feuilleter à l'aide de variables et des arguments état et/ou échéance, on dispose de différents messages pour déclencher l'exécution de cette opération.

a) **feuilleter:** uneVariable **ordreFeuilletage:** unOrdre **bureau:** unBureau.

b) **feuilleter:** uneVariable **état:** unEtat **échéance:** uneEchéance **ordreFeuilletage:** unOrdre **bureau:** unBureau.

c) **feuilleter:** unNom **type:** unType **ordreFeuilletage:** unOrdre **bureau:** unBureau.

d) **feuilleter:** unNom **type:** unType **état:** unEtat **échéance:** uneEchéance **ordreFeuilletage:** unOrdre **bureau:** unBureau.

où

* unNom, unType, unEtat, unOrdre sont des instances de la classe Symbol (point 4.1.2.2).
* unBureau est une instance de la classe Bureau.
* unOrdre prend une des valeurs suivantes: #fifo, #lifo, #alphabétique, #alphabétiqueInverse, #aucun (en minuscules et/ou majuscules).
* uneVariable est une variable (point 6.6.1.1) et qui a reçu une valeur grâce à l'exécution d' une opération enlever ou grâce à l'utilisation du composant de type boucle.

° Précondition

- L'objet informationnel spécifié existe dans l'environnement du bureau unBureau et est structurant.

° Postconditions

- L'objet informationnel spécifié a été feuilleté et est associé à la table de travail. Si des constituants de l'objet sont associés à un autre objet de travail, ils ne sont pas feuilletés.
- Les valeurs, différentes de nil, des arguments unEtat et/ou uneEcheance sont affectées aux variables d'instance de l'objet informationnel.

° Exemple

Dossier feuilleter: #étudiant
type: #typeEtudiant
ordreFeuilletage: #aucun
bureau: b.

L'exécution de cette expression permet de feuilleter le dossier de nom #étudiant et de type # typeEtudiant dans le bureau b.

6.6.2.17 L'opération **enlever**

° Cette opération permet d'enlever le premier élément d'une pile et de le placer sur le bureau. Selon la manière utilisée pour désigner la pile, on dispose de messages distincts pour déclencher l'exécution de cette opération:

a) **enleverDe:** unPileNom **bureau:** unBureau
b) **enleverDe:** unevariable **bureau:** unBureau

où

* unPileNom est une instance de la classe Symbol (point 4.1.2.2).
* unBureau est une instance de la classe Bureau.
* uneVariable est une variable au sens où nous l'avons défini au point 6.6.1.1 et qui a reçu une valeur grâce à l'exécution d' une opération enlever ou grâce à l'utilisation du composant de type boucle.
* unBureau est une instance de la classe Bureau.

° Préconditions

* La pile spécifiée existe dans l'environnement du bureau unBureau et n'est pas vide. Une pile est vide si physiquement, elle ne possède aucun constituant. L'analyste aura à sa charge la vérification de l'état d'une pile, il le fera par l'intermédiaire de messages permettant de consulter la valeur de la variable d'instance *etatContenu.*
* le premier élément de la pile ne lui est plus associé mais est associé à la table de travail.
* la méthode activée par la réception de ce message renvoie une variable (point 6.6.1.1) contenant l'identifiant de l'objet enlevé de la pile.

° Exemple

information <- Pile enleverDe: #infoEnAttente
bureau: b.

L'exécution de cette expression permet d'enlever l'objet informationnel se trouvant au sommet de la pile de nom #infoEnAttente figurant dans le bureau b. L'identifiant de cet objet sera affecté à la variable information.

6.6.2.18 L'opération **vérifier**

° Cette opération permet de vérifier la complétude d'un dossier ou d'un formulaire. Un dossier est jugé complet si toutes les pièces du dossier s'y trouvent. Un formulaire est complet si tous les champs sont remplis. Plusieurs messages sont disponibles pour déclencher l'exécution de cette opération:

a) **vérifier:** uneVariable **bureau:** unBureau
b) **vérifier:** uneVariable **état:** unEtat **échéance:** uneEcheance **bureau:** unBureau
c) **vérifier:** unNom **numCopie:** unNumCopie **type:** unType **bureau:** unBureau
d) **vérifier:** unNom **numCopie:** unNumCopie **type:** unType **état:** unEtat **échéance:** uneEchéance **bureau:** unBureau

où

* unNom, unType, unEtat sont des instances de la classe Symbol (point 4.1.2.2).
* uneEcheance est une instance de la classe String (point 4.1.2.2) de la forme 'MM/JJ/AAAA'.
* unNumCopie est une instance de la classe SmallInteger (point 4.1.2.1).
* uneVariable est une variable (point 6.6.1.1) qui a reçu une valeur grâce à l'exécution d'une opération enlever ou grâce à l'utilisation du composant de type boucle.
* unBureau est une instance de la classe Bureau.

° Précondition

- L'objet informationel spécifié est un formulaire ou un dossier et appartenir à l'environnement du bureau unBureau.

° Postconditions

- L'objet informationnel est associé à la table de travail.
- La vérification est faite et la méthode activée renvoit le résultat de la vérification. Il peut donc être affecté à une variable Smalltalk et ne peut prendre qu'une des deux valeurs suivantes: #COMPLET, #INCOMPLET.
- Les valeurs, différentes de nil, des arguments unEtat et/ou uneEcheance sont affectées aux variables d'instance de l'objet informationnel.

° exemple

```
paramètre <- Formulaire vérifier: #étudiant
                        numCopie: 3
                        type: #inscription
                        bureau: b.
```

L'exécution de cette expression affecte à la variable Smalltalk paramètre le résultat de la vérification du formulaire spécifié par les arguments #étudiant, 3,

#inscription.

6.6.2.19 L'opération **trier**

° Cette opération permet de modifier l'ordre de classement d'un objet informationnel structurant. L'ordre de classement des constituants d'un objet informationnel structurant est spécifié lors de la création de l' objet informationnel structurant. Nous prendrons en compte 5 ordres différents pour les objets informationnels structurants: LIFO, FIFO, ALPHABETIQUE, ALPHABETIQUEINVERSE, AUCUN. Pour la pile, seul l'ordre de classement LIFO est possible. Selon les arguments désirés, on disposera, pour déclencher l'exécution de cette opération, des messages suivants:

a) **trier:** uneVariable **état:** unEtat **échéance:** uneEchéance
ordreTri: unOrdreTri **bureau:** unBureau
b) **trier:** uneVariable **ordreTri:** unOrdreTri **bureau:** unBureau
c) **trier:** unNom **type:** unType **état:** unEtat **échéance:** uneEchéance
ordreTri: unOrdreTri **bureau:** unBureau
d) **trier:** unNom **type:** unType **ordreTri:** unOrdreTri **bureau:** unBureau

où

* unNom, unType, unEtat, unOrdreTri sont des instances de la classe Symbol (point 4.1.2.2).
* unOrdreTri prend les valeurs suivantes: #LIFO, #FIFO, #ALPHABETIQUE, #ALPHABETIQUEINVERSE, #AUCUN.
* uneEchéance est une instance de la classe String (point 4.1.2.2) de la forme 'MM/JJ/AAAA'.
* unNumCopie est une instance de la classe SmallInteger (point 4.1.2.1).
* uneVariable est une variable au sens où nous l'avons défini au point 6.6.1.1 et qui a reçu une valeur grâce à l'exécution d' une opération enlever ou grâce à l'utilisation du composant de type boucle.
* unBureau est une instance de la classe Bureau.

° Précondition

* L'objet informationnel spécifié existe dans l'environnement de bureau et est un objet informationnel structurant.
* unOrdreTri prend une de ces valeurs suivantes: #LIFO, #FIFO, #ALPHABETIQUE, #ALPHABETIQUEINVERSE, #AUCUN.

° Postconditions

* Les constituants de l'objet informationnel strucurants sont classés selon le nouvel ordre.
* Dans le cas du dossier et du fichier, l'ordre de tri devient le nouvel ordre de classement. Dans le cas du tri d'une pile, les éléments de la pile sont triés selon l'ordre spécifié; mais lors de l'insertion d'un élément dans la pile, celui-ci est inséré selon l'ordre LIFO. On peut donc trier les éléments d'une pile mais l'ordre de classement reste LIFO.

- Les valeurs, différentes de nil, des arguments unEtat et/ou uneEcheance sont affectées aux variables d'instance de l'objet informationnel.

° Exemple:

Fichier trier: #inscription
type: #typeInscription
etat: #trié
échéance: '12/25/1988'
ordreTri: #FIFO
bureau: b.

Cette expression permet de trier, dans le bureau b, le fichier spécifié selon l'ordre FIFO.

6.6.2.20 L'opération **reproduire**

° Cette opération permet de reproduire, en plusieurs exemplaires, un objet informationnel élémentaire. Il ne sera pas permis de reproduire les objets informationnels structurants, à moins de reproduire chacun de leurs constituants.Deux messages sont disponibles afin de réaliser cette opération:

a) **reproduire:** unNom **numCopie:** unNumCopie **type:** unType **quantité:** unNombre1 **àPartirDe:** unNombre2 **bureau:** unBureau
b) **reproduire:** uneVariable **quantité:** unNombre1 **àPartirDe:** unNombre2 **bureau:** unBureau

où

* unNom et unType sont des instances de la classe Symbol (point 4.1.2.2).
* unNumCopie, unNombre1 et unNombre2 sont des instances de la classe SmallInteger (point 4.1.2.1).
* uneVariable est une variable (point 6.6.1.1) qui a reçu une valeur grâce à l'exécution d' une opération enlever ou grâce à l'utilisation du composant de type boucle.
* unBureau est une instance de la classe Bureau.

° Préconditions

* L'objet informationnel spécifié existe dans l'environnement du bureau unBureau et est un objet informationel élémentaire.
* Une photocopieuse doit exister dans l'environnement du bureau unBureau.
* Le numéro de départ, unNombre2 est supérieur au numéro des copies déjà existantes pour l'original reproduit; même si on fait des copies d'une copie.

° Postconditions

* Les copies sont créées et associées à la photocopieuse du bureau

unBureau.
* L'original copié ou la copie copiée est associée à la photocopieuse du bureau unBureau.

° En outre, si unNombre1 a reçu la valeur **nil**, alors on reproduit à un seul exemplaire. Si le nombre unNombre2, spécifiant le numéro de départ pour les copies est **nil**, les nouvelles copies sont numérotées à partir du dernier numéro, augmenté de 1, affecté à une copie de l'original. Hormis le numéro de copie, chaque copie a les mêmes valeurs d'attributs que l'objet qui a servi à la copie.

° Exemple:

```
Formulaire reproduire: #étudiant
                numCopie: 0
                type: #inscription
                quantité: 2
                àPartirDe: 1
                 bureau: b.
```

L'exécution de cette expression permet de reproduire en deux exemplaires, le formulaire de nom #étudiant, de numéro de copie 0 et de type #inscription. Les nouvelles copies seront numérotées à partir de 1.

Nous venons de passer en revue les différentes opérations primitives observables dans un bureau (point 2.3) et implémentées dans notre programme. Cependant, Les opérations primitives ne suffisent pas pour décrire une täche. Il s'avère en outre nécessaire d'introduire un enchaînement entre les opérations. Dans ce paragraphe, nous allons décrire les structures d'enchaînements utilisés.

6.6.3 Les structures d'enchaînement

Les structures d'enchaînement que nous allons décrire font partie du langage de spécification détaillé au chapitre 2 (point 2.4.4.5 et 2.4.4.6). Comme le reste de notre logiciel, elles ont également été implémentées en Smalltalk-80 et reposent donc sur les mêmes principes que les structures de contrôles (point 3.4.4) de ce même langage, à savoir l'envoi de messages à des objets.

Rappelons que, dans le langage de spécification, existent les structures d'enchaînement suivantes:

- le test de la valeur d'un attribut
- le test d'appartenance d'un objet informationnel à un autre objet informationnel
- la boucle

6.6.3.1 Le test de la valeur d'un attribut

° Ce test permet de tester la valeur d'un attribut d'un objet informationnel

spécifié directement ou référencé par une variable. En fonction de la valeur du test, on exécute telle ou telle suite d'expressions. Le test est de la même forme que la sélection conditionnelle Smalltalk exposée au point 3.4.4.2 et consiste à comparer la valeur d'un attribut - une variable d'instance - à un autre objet, essentiellement une instance de la classe Symbol (point 4.1.2.2), String (point 4.1.2.2) ou SmallInteger (point 4.1.2.1).

Les opérateurs de comparaison utilisés sont les suivants: {**=**, **=not**, **<**, **>**, **<=**, **>=** }.

° Afin d'établir la valeur d'un attribut d'un objet informationnel, on a recours à des messages implémentés dans le protocole de classe de la classe Information. Il peuvent donc être envoyés à cette classe ou à une de ses sous-classes.

Suivant que l'on spécifie l'objet informationnel à l'aide d'une variable - au sens défini au point 6.6.1.1 - ou des arguments classiques, on dispose de deux messages distincts pour déclencher le test de la valeur d'un attribut:

1) **valeurAttributDe:** uneVariable **attribut:** unAttribut **bureau:** unBureau
2) **valeurAttributDe:** unNom **type:** unType **numCopie:** unNumCopie **attribut:** unAttribut **bureau:** unBureau

où

* unNom, unType, unAttribut sont des instances de la classe Symbol (point 4.1.2.2).
* unNumCopie est une instance de la classe SmallInteger (point 4.1.2.1).
* unBureau est une instance de la classe Bureau.
* uneVariable est une variable (point 6.6.1.1) qui a reçu une valeur grâce à l'exécution d' une opération enlever ou grâce à l'utilisation du composant de type boucle.
* unAttribut prend les valeurs suivantes:

```
#nom
#type
#numCopie
#état
#échéance
#classe.
```

° Exemple

```
(Information valeurAttributDe: variable
                    attribut: #état
                    bureau: b) = #vérifié
    ifTrue:[ Information replacer: variable bureau: b ]
    ifFalse:[ parametre <- Information vérifier: variable bureau: b ]
```

L'exécution de cette suite d'expressions permet, dans le bureau b, d'obtenir la

valeur de l'attribut *état* de l'objet informationnel identifié par la variable *variable* et de comparer cette valeur au symbole #vérifié. Si la valeur du test est true, on replace l'objet informationnel dans son lieu de rangement ou de classement, sinon on en vérifie la complétude.

6.6.3.2 Le Test d'appartenance d'un objet informationnel à un autre objet informationnel

° Ce test permet, en fonction de l'appartenance ou non d'un objet informationnel à un autre, d'exécuter telle ou telle suite d'expressions. Ces objets informationnels peuvent être désignés à l'aide de variables ou d' arguments classiques. Ce test a également la forme de la sélection conditionnelle classique en Smalltalk exposée au point 3.4.4.2 et utilise le message **ifTrue: ifFalse:** . Ce message est envoyé à une expression dont l'exécution permet de savoir si un objet informationnel appartient à un autre.

° Cette expression est rédigée à l'aide des messages implémentés dans le protocole de classe de la classe Information. Ils peuvent donc être envoyés à une de ses sous-classes.

Selon la manière dont on veut désigner les objets, les messages suivants sont disponibles:

1) **appartenanceDe:** uneVariable1 **a:** uneVariable2 **bureau:** unBureau
2) **appartenanceDe:** uneVariable1 **a:** uneOisClasseNom **oisNom:** unOisNom **oisType:** unOisType **bureau:** unBureau
3) **appartenanceDe:** unInfoNom **numCopie:** unInfoNumCopie **type:** unInfoType **a:** uneOisClasseNom **oisNom:**unOisNom **oisType:** unOisType **bureau:** unBureau
4) **appartenanceDe:** unInfoNom **numCopie:** unInfoNumCopie **type:** unInfoType **a:** uneVariable2 **bureau:** unBureau

où

* uneVariable1 et uneVariable2 sont des variables (point 6.6.1.1) qui ont reçu une valeur grâce à l'exécution d'une opération enlever ou grâce à l'utilisation du composant de type boucle.
* unInfoNumCopie est une instance de la classe SmallInteger (point 4.1.2.1).
* unBureau est une instance de la classe Bureau.
* uneOisClasseNom, unOisNom, unOisType, unInfoNom et unInfoType sont des instances de la classe Symbol (point 4.1.2.2).
* uneOisClasseNom désigne le nom d'une sous-classe de la classe InfoStructurant.

Chacune des méthodes activées par ces différents messages renvoit true si le premier objet désigné dans le message appartient au second, et false dans le cas contraire.

° Exemple

```
(Formulaire appartenanceDe: #étudiant
                numCopie: 0
                type: #typeEtudiant
                a: variable
                bureau:b)

  ifTrue:[ Formulaire replacer: #étudiant numCopie: 0 type:
                #typeEtudiant bureau: b ]
  ifFalse:[ Formulaire compléter: #étudiant numCopie: 0 type:
                #typeEtudiant bureau: b ]
```

L'exécution de ces expressions permettent de tester si un formulaire spécifié appartient à l'objet informationnel spécifié par la variable *variable*.. Si c'est le cas, on replace le formulaire dans son lieu de rangement ou de classement, sinon on le complète.

6.6.3.3 La boucle

° Cette structure d'enchaînement permet de sélectionner un sous-ensemble d'objets informationnels dans un bureau donné afin de leur appliquer un certain traitement.

° Le principe est de sélectionner un ensemble d'objets informationnels sous forme d'une collection de variables (point 6.6.1.1) - chaque variable identifiant un objet sélectionné - et envoyer le message **do:** (point 3.4.4.4). L'argument de ce message est un bloc paramétré, le paramètre étant destiné à contenir la variable courante de la collection. Il contient la suite d'expressions à exécuter pour chaque objet sélectionné.La boucle est donc, après l'opération enlever, la deuxième manière d'utiliser une variable.

° Les messages à utiliser pour sélectionner les objets informationnels ont été implémentés dans le protocole de classe de la classe Information et peuvent donc être envoyés à chacune des sous-classes de cette classe. Il s'agit de:

1) La sélection des constituants d'un objet informationnel structurant

Selon que l'on désire spécifier l'objet informationnel structurant à l'aide d'une variable ou non, on dispose de deux messages:

a) **tous:** unSymbôle **de:** uneVariable **dans:** unBureau
b) **tous:** unSymbôle **de:** unOisClasseNom **nom:** unOisNom **type:** unOisType
   dans: unBureau

où

* unSymbôle, unOisClasseNom, unOisNom et unOisType sont des

instances de la classe Symbol (point 4.1.2.2).
* unBureau est une instance de la classe bureau.
* unSymbôle prend une des valeurs suivantes:

- #TOUS
- #OIE
- #OIS

Ces deux messages, envoyés à la classe Information, renvoie une collection de variables contenant les identifiants de tous les objets informationnels, de tous les objets informationnels élémentaires ou de tous les objets informationnels structurants, et ce suivant la valeur de unSymbôle, appartenant à l'objet informationnel structurant spécifié dans le message. L'objet informationnel structurant doit appartenir au bureau unBureau.

2) La sélection de tous les objets informationnels d'une classe donnée et appartenant à un bureau donné

Le message, à envoyer à la classe Information, est le suivant:

tous: unClasseNom **dans:** unBureau

où

* unClasseNom est une instance de la classe Symbol (point 4.1.2.2) désignent le nom d'une sous-classe de la classe InfoElementaire ou InfoStructurant.
* unBureau est une instance de la classe Bureau.

Ce message, envoyé à la classe Information, permet d'obtenir une collection de variables contenant les identifiants de toutes les instances de la classe dont le nom est désigné par unClasseNom et qui appartiennent au bureau unBureau.

3) La sélection de tous les objets informationnels d'un bureau

Le message, à envoyer à la classe Information, est le suivant:

toutesInfoDans: unBureau

L'envoi de ce message à la classe Information renvoie une collection des identifiants, sous forme de variables, de tous les objets informationnels du bureau unBureau.

° Exemple

```
(Information tous: #OIE
              de: #Dossier
              nom: #étudiant
```

```
                type: #typeinscription
                dans: b)
    do: [:variable | Information reproduire: variable
                            quantité: 1
                            àPartirDe: 5
                            bureau: b]
```

L'exécution de cette suite d'expressions permet de sélectionner tous les objets informationnels élémentaires classés dans le dossier spécifié et de les reproduire en un exemplaire.

Les structures d'enchaînement que nous venons de passer en revue, représentent l'implémentation Smalltalk de celles définies dans le langage de spécification de [VPMS-87].

On peut trouver des exemples d'application plus détaillés de ces structures d'enchaînement dans l'annexe 2 de ce mémoire.

6.7 Conclusion

Dans ce chapitre, nous avons déterminé les classes dont les instances doivent nous servir de modèles dans notre programme. Nous avons organisé ces classes en hiérarchie en utilisant au maximum le principe d'abstraction de manière à profiter de l'héritage propre à la programmation orientée objet et à Smalltalk. Ensuite, nous avons recensé les variables qui nous permettent de mémoriser l'état de nos modèles. Enfin, nous avons passé en revue une partie de leur interface, à savoir les messages utilisés par l'analyste pour décrire un environnement et des tâches de bureau.

Remarquons que, pour un analyste non habitué au langage Smalltalk, certains messages peuvent paraître longs et lourds à manipuler. En effet, ces messages pêchent souvent par leur longueur et par le fait que pour un message donné, tous les arguments doivent être spécifiés. La valeur **nil** peut cependant leur être affectée. La raison de ceci est que les messages et les méthodes qui leur sont associées ont été implémentés en fonction d'un interface futur entre le langage Smalltalk d'une part et le langage de spécification [VPMS-87] exposé brièvement au chapitre 2 d'autre part. Cet interface formerait une couche supérieure, au dessus du langage Smalltalk, qui permettrait de cacher les inconvénients présentés par les messages qui vont être exposés. C'est ainsi que des arguments obligatoires en Smalltalk deviendraient facultatifs, comme ceux décrits dans le langage de haut niveau. Ceci devrait permettre d'alléger la tâche de l'analyste dans sa description du bureau et des tâches.

Dans le chapitre suivant, nous allons examiner les classes qui doivent nous permettre d'associer une vue et un contrôleur à chacun de nos modèles.

CHAPITRE 7

LES VUES ET CONTROLEURS DU LOGICIEL

CHAPITRE 7: LES VUES ET CONTROLEURS DU LOGICIEL

Comme nous l'avons dit au chapitre 4, l'ensemble de l'environnement de programmation Smalltalk-80 est entièrement organisé autour d'une architecture à trois couches appelée Modèle-Vue-Contrôleur. En outre, le développement de tout programme à vocation graphique dans cet environnement doit se baser sur cette architecture. Ceci implique généralement la création de classes spécifiques qui s'insèrent dans la hiérarchie des classes du système Smalltalk.

Au chapitre 6, nous avons déterminé nos modèles, nous allons maintenant décrire les vues permettant de présenter le bureau ainsi que ses composants et d'exécuter les simulations de tâches à l'écran. Nous décrirons également les contrôleurs chargés de gérer la communication avec les utilisateurs.

7.1 Les vues

L'objectif de ce mémoire est de concevoir un outil d'aide à l'automatisation du travail de bureau dont le principe est la simulation graphique des tâches. Afin de réaliser cette simulation, nous devons présenter deux types de données à l'écran. Ces données sont d'une part le bureau et d'autre part les objets informationnels, de rangement, de travail, et d'interface qui le composent. Le bureau est présenté à l'écran dans une fenêtre. Cette dernière est destinée à contenir le décor du bureau, mais c'est également dans cette fenêtre que seront affichés les différents objets appartenant au bureau et que se dérouleront les simulations de tâches. Quant aux objets affichés dans la fenêtre du bureau, ils sont présentés chacun dans une fenêtre.

Comme nous l'avons dit au chapitre 4, l'affichage d'informations dans des fenêtres utilise un mécanisme intermédiaire appelé *vue* . En Smalltalk, une vue est également un objet et est donc décrite dans une classe. On trouve dans le système Smalltalk-80 de nombreuses classes de ce type telles que FormView, ListView, ... ; chacune d'entre elles variant selon le type d'information à présenter. Toutes ces classes possèdent la classe View dans la chaîne de leurs super-classes. C'est dans cette classe que sont définies toutes les structures de données et les méthodes communes à l'ensemble des vues.

Afin de gérer l'affichage de nos données, deux nouvelles classes ont du être créées: la classe ComposantView et la classe BureauView.

7.1.1 La classe ComposantView

Les objets à afficher sont des objets de bureau et constituent la majeure partie des modèles de notre programme. Ces objets sont des instances des classes décrites au chapitre 6, c'est-à-dire des sous-classes des classes InfoElementaire, InfoStructurant, Rangement, Travail et Interface. Chacune de ces classes possède la classe composant dans la chaîne de ses super-classes, cette classe fournit la description des variables et messages communs à tous les objets de bureau.

En ce qui concerne l'affichage d'un objet de bureau, les structures de données et

les méthodes nécessaires sont identiques quelle que soit la classe de cet objet. C'est la raison pour laquelle nous n'avons pas été obligé de créer une classe particulière pour décrire les vues associées à chaque type d'objets et que l'on peut se contenter d'une seule classe appelée ComposantView. Ceci nous évite de définir une multitude de classes EtagereView, ArmoireView, ... qui ne posséderaient de toutes façons que des variables et des messages similaires.

Chaque objet à présenter à l'écran est associé à une instance de la classe ComposantView dont il constitue le modèle. Entre autres informations, ce modèle fournit le dessin à afficher.

En Smalltalk, toute nouvelle classe doit s'insérer dans la hiérarchie du système. La classe ComposantView a la classe View comme super-classe (figure 7.1). En effet, notre nouvelle classe ne se base que sur des variables et méthodes communes à toutes les vues du système. En outre, la classe ComposantView definit de nouveaux messages propres à l'affichage d'objets de bureau et à la coopération du modèle avec sa vue.

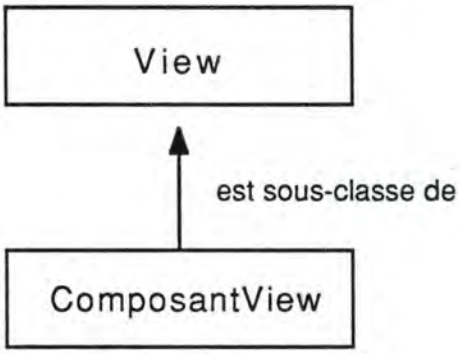


figure 7.1: La classe ComposantView et sa super-classe

Parmi les variables héritées de la classe View, on trouve la variable d'instance *model* dans laquelle est rangée la référence au modèle. Une instance de la classe ComposantView possède une référence explicite vers son modèle. Un exemple illustrant cette relation entre le modèle et sa vue est présentée à la figure 7.2. Dans cet exemple, uneComposantView, instance de la classe ComposantView est associée à une instance uneEtagere de la classe Etagere grâce à la variable d'instance *model*. L'instance uneComposantView possède une référence vers son modèle uneEtagere, l'inverse n'étant pes le cas.

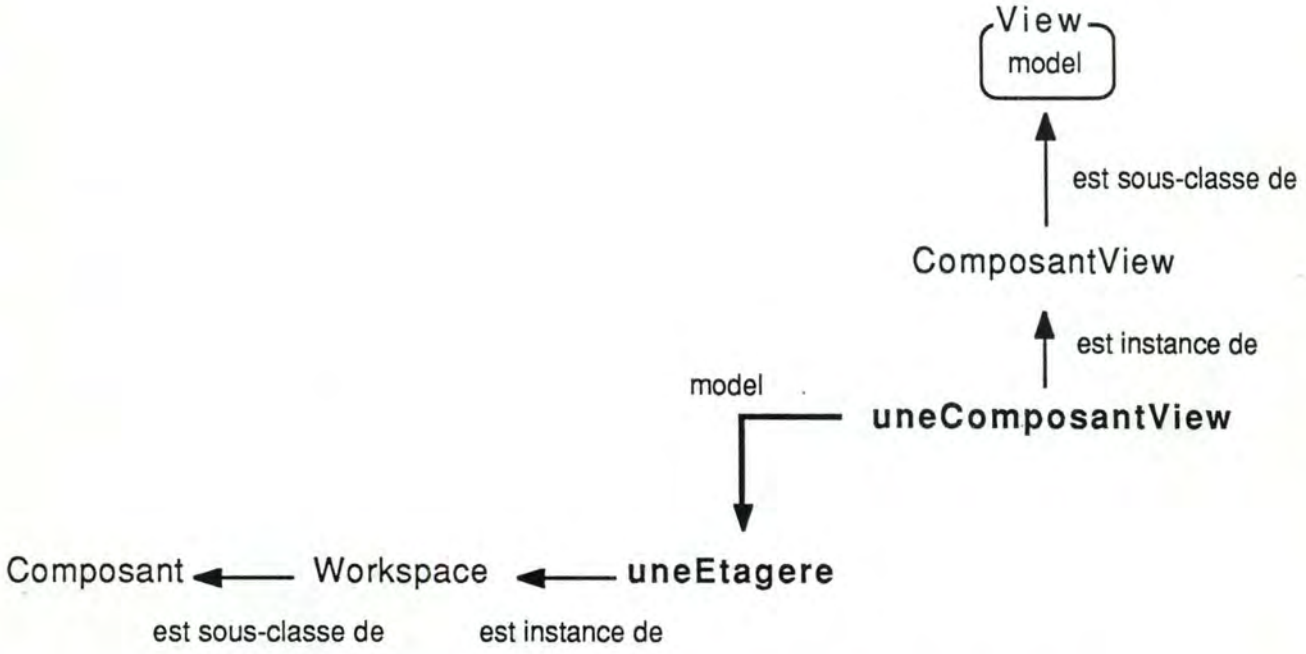


figure 7.2: Les relations existants entre un objet de bureau et sa vue

7.1.2 La classe BureauView

Le mécanisme appliqué pour l'affichage du bureau dans une fenêtre suit le même principe que celui utilisé dans le cas des objets de bureau. En effet, le bureau, une instance de la classe Bureau fournit les données à une vue qui se charge de les présenter à l'écran. Cette vue est décrite par la classe BureauView.

Cette nouvelle classe a la classe View comme super-classe (figure 7.3), les instances de la classe ComposantView n'ont en effet besoin que des variables et méthodes de base communes à toutes les vues.

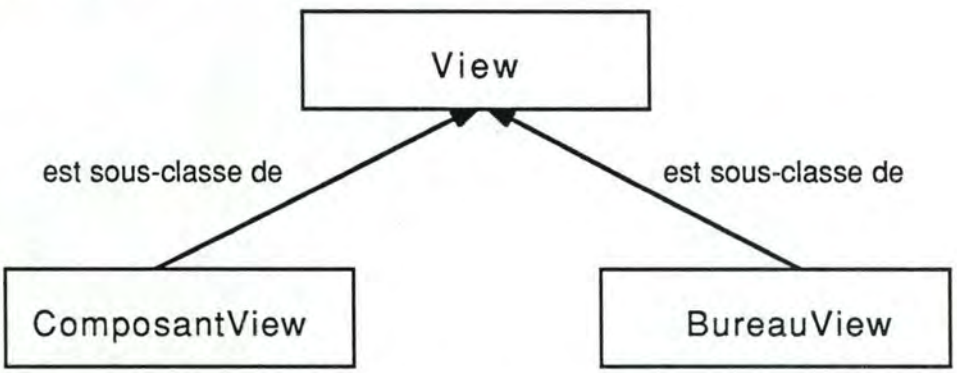


figure 7.3: La classe BureauView et sa super-classe

En outre, cette classe définit de nouvelles méthodes nécessaires à la coopération de la vue avec son modèle - en l'occurrence le bureau. Ces méthodes permettent notamment d'afficher le décor du bureau mais aussi les différentes fenêtres contenant chacune un objet du bureau.

Chaque instance de la classe BureauView hérite, de la classe View, la variable d'instance *model* dans laquelle se trouve la référence au modèle. La figure 7.4 illustre cette relation entre le bureau et sa vue.

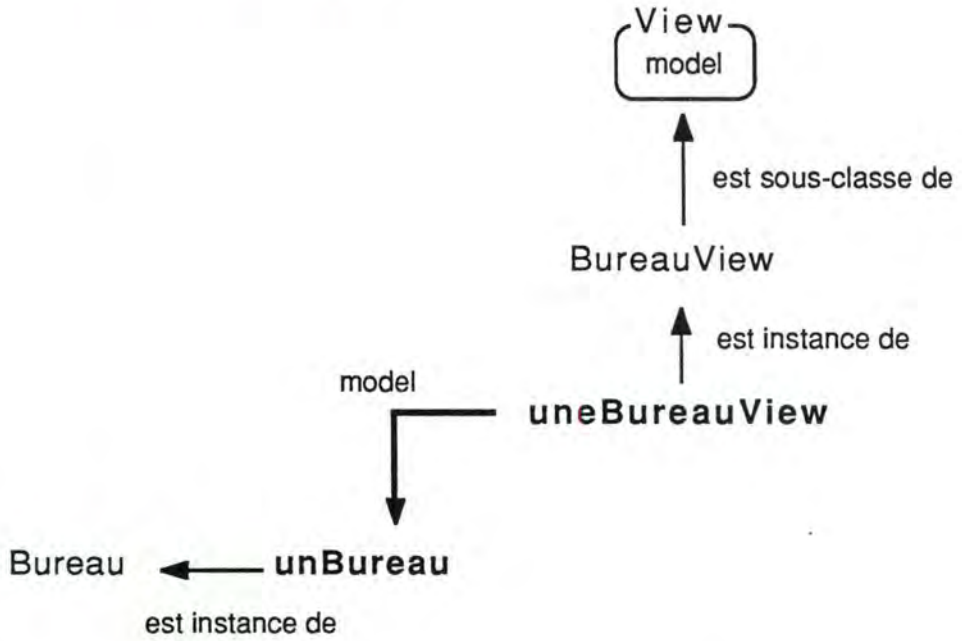


figure 7.4: Les relations existants entre un bureau et sa vue

7.2 Les contrôleurs

Jusqu'à présent, tout ce que nous avons réalisé à l'aide des modèles et vues, c'est l'affichage à l'écran d'un bureau et de ses objets. Cependant aucune communication n'est encore rendue possible entre ceux-ci et un utilisateur potentiel.

Il serait en effet interessant qu'un utilisateur puissent manipuler les fenêtres et les objets qu'elles contiennnent à l'aide soit du clavier soit de la souris. Il pourrait ainsi leur appliquer des opérations, celles-ci pouvant être tout à fait générales telles que fermer, agrandir ou encore déplacer une fenêtre mais aussi plus spécifiques telles qu'obtenir des informations concernant un objet de bureau.

Cette communication est gérée grâce à des objets appelés *contrôleurs* (point 4.2). Ces objets sont chargés de détecter les commandes de l'utilisateur et de déclencher les actions adéquates. Comme tout objet Smalltalk, un contrôleur est décrit dans une classe (MouseMenuController, StandardSystemController, ...) qui définit un type de contrôleur particulier. Chacune de ces classes possède la classe Controller dans la chaîne de ses super-classes dont elle hérite toutes les propriétés.

Afin de répondre à nos besoins, deux nouvelles classes ont été ajoutées à la hiérarchie existante. La classe ComposantController et BureauController.

7.2.1 La classe ComposantController

La classe ComposantController fut créée pour permettre à l'utilisateur d'agir sur les objets du bureau. En effet, chaque instance de la classe ComposantView est associée à un contrôleur, instance de la classe ComposantController.

Comme chaque objet de bureau est instance d'une sous-classe de Composant et comme chaque objet présenté à l'écran est associé à une instance de ComposantView, une seule classe ComposantController a été créée pour gérer les interactions avec l'utilisateur.

La classe ComposantController a la classe MouseMenuController pour super-classe. La classe Controller étant la super-classe de cette dernière (figure 7.5).

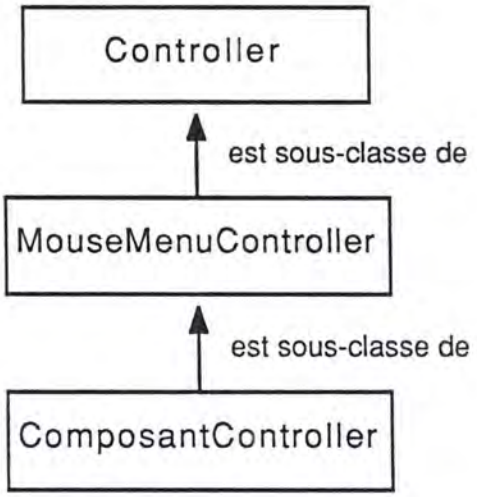


figure 7.5: La classe ComposantController et ses super-classes

La classe ComposantController définit les structures de données et les méthodes communes à tous les contrôleurs du système.

Quant à la classe MouseMenuController, elle contient toutes les variables et méthodes nécessaires à la création de contrôleurs permettant de gérer une fénêtre à l'aide de la souris. Elle offre notamment la possibilité de créer des Pop-Up menus associés aux diiférents boutons de la souris. Lors de la création d'une de ses instances, ou d'une instance d'une de ses sous-classes, elle associe automatiquement au nouveau contrôleur un Pop-Up menu accessible grâce au bouton de droite de la souris. Ce menu comprend des options communes à tous les contrôleurs de ce type: fermer, déplacer, agrandir, ... la fenêtre à laquelle il est associé. Toutefois, ce menu peut-être redéfini dans une des sous-classes.

Dans la classe ComposantController, outre les variables et les méthodes héritées de ses super-classes, on trouve de nouvelles méthodes nécessaires à la coopération du contrôleur avec son modèle et la vue qu'il contrôle.

Chaque instance de la classe ComposantController hérite, de la classe Controller, la variable d'instance *model* dans laquelle se trouve la référence au modèle et la variable d'instance *view* dans laquelle se trouve la référence à la vue.

Dans ce cas-ci, le modèle est défini par la classe Composant et la vue par la classe ComposantView. La figure 7.6 complète l'exemple du point 7.1.1 et illustre les relations entre les trois composants: modèle, vue et contrôleur. Dans ce schéma, nous avons également ajouté la variable *controller* dont héritent toutes les instances de la classe ComposantView. On y aperçoit que la vue uneComposantView possède une référence vers son modèle et vers son contrôleur, que le contrôleur unComposantController possède une référence vers son modèle et vers sa vue mais que le modèle est tout à fait indépendant de sa vue et de son contrôleur.

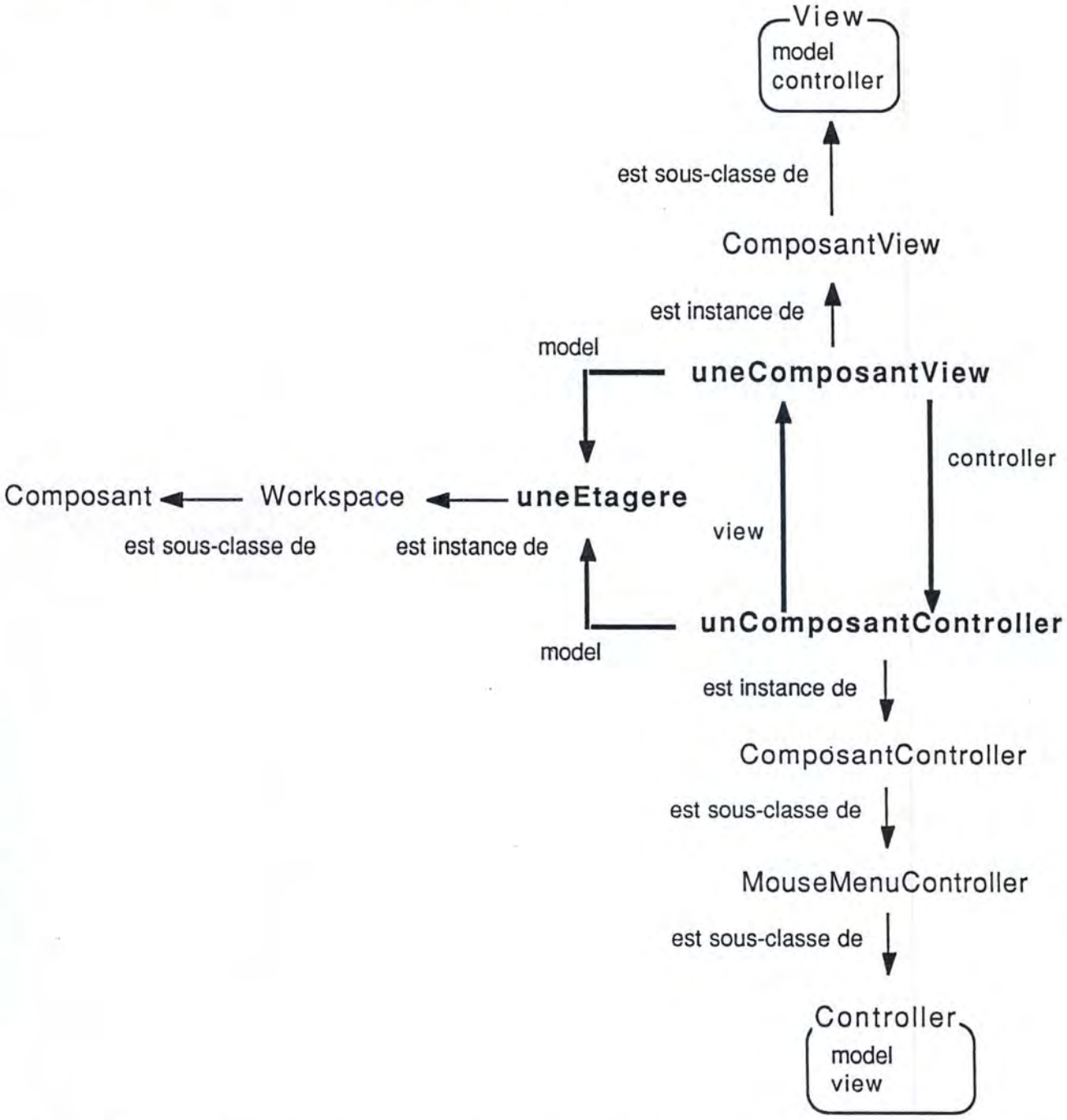


figure 7.6: Les relations existants entre un objet de bureau, sa vue et son contrôleur

7.2.2 La classe BureauController

Comme nous l'avons exposé précédemment (point 7.1), nous avons besoin d'une vue pour gérer la simulation graphique se déroulant dans la fenêtre du bureau. Cette vue doit également servir à afficher le décor du bureau à automatiser. BureauView décrit l'implémentation d'une telle vue. A la vue du bureau est associée un contrôleur permettant à l'analyste de communiquer avec le bureau. Ce contrôleur est décrit par la classe BureauController.

Comme la classe ComposantController, la classe BureauController a la classe MouseMenuController pour super-classe (figure 7.7).

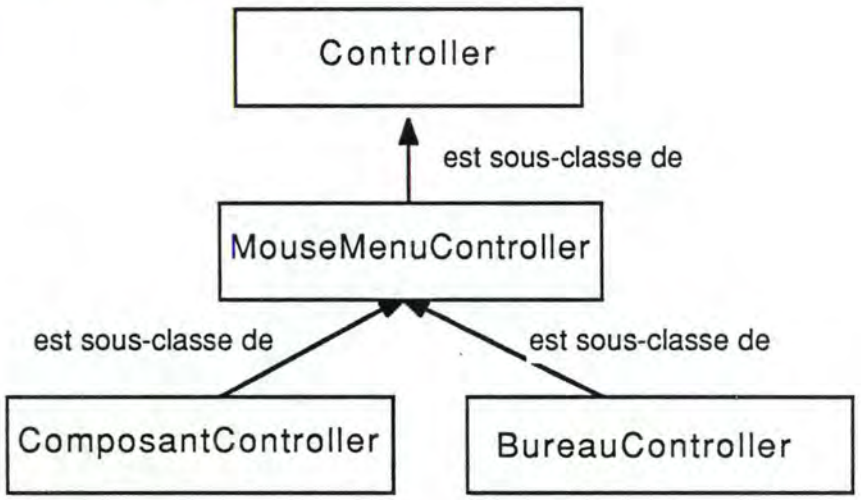


figure 7.7: La classe BureauController et ses super-classes

Outre les variables et méthodes héritées de ses super-classes, la classe BureauController définit de nouvelles méthodes nécessaires à la coopération du contrôleur avec son modèle et la vue qu'il contrôle.

Chaque instance de la classe BureauController hérite, de la classe Controller, la variable d'instance *model* dans laquelle se trouve la référence au modèle et la variable d'instance *view* dans laquelle se trouve la référence à la vue. Dans ce cas-ci, le modèle est défini par la classe Bureau et la vue par la classe BureauView. La figure 7.8 complète la figure 7.4 et illustre les relations entre le modèle, sa vue et son controller. Dans ce schéma, nous avons également inséré la variable *controller* déclarée dans la classe View et dont hérite la classe BureauView. On y aperçoit que la vue uneBureauView possède une référence vers son modèle et vers son contrôleur, que le contrôleur unBureauController possède une référence vers son modèle et vers sa vue mais que le modèle est tout à fait indépendant des deux autres composants de l'architecture.

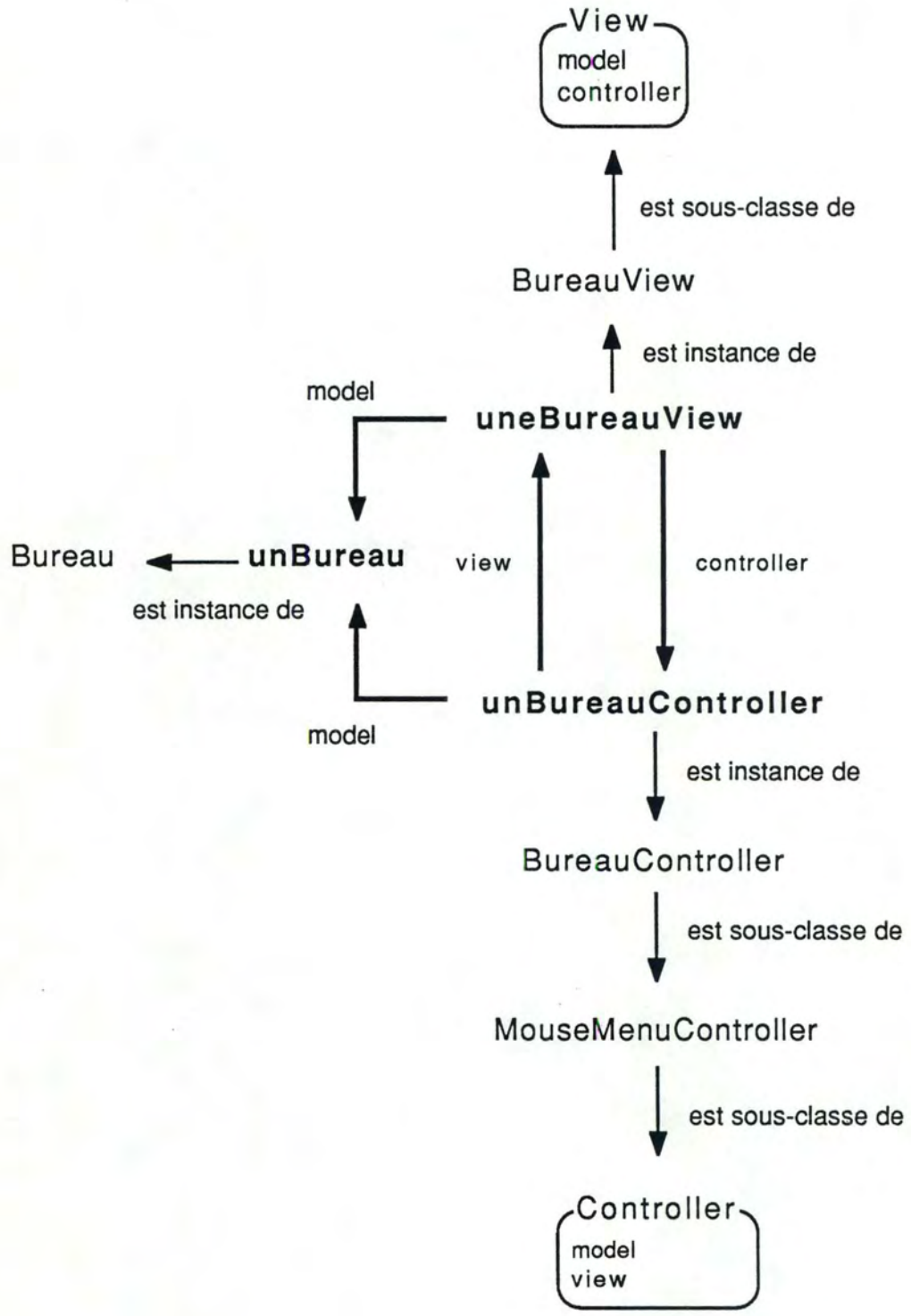


figure 7.8: Les relations existants entre un bureau, sa vue et son contrôleur

7.3 Conclusion

Dans le chapitre 4, nous avons exposé l'architecture autour de laquelle s'articule l'ensemble du système Smalltalk-80.

Au chapitre 5, nous avons décrit les grandes lignes de l'architecture de notre logiciel et montré que celle-ci repose et s'insère dans l'architecture du système Smalltalk.

Dans le chapitre 6, nous avons décrit l'ensemble des classes composant la première couche de notre architecture à savoir les modèles.

Enfin, dans le chapitre 7, nous avons montré quels étaient nos vues et contrôleurs.

Dans le chapitre qui va suivre, nous allons nous attacher à montrer en quoi l'architecture choisie présente bien des avantages et permet de combler de nombreuses lacunes du prototype réalisé l'an dernier.

CHAPITRE 8

LES AVANTAGES DE SMALLTALK-80 ET DE SON ARCHITECTURE

CHAPITRE 8: LES AVANTAGES DE SMALLTALK-80 ET DE SON ARCHITECTURE

Comme cela a été dit au chapitre 2 (point 2.5.2), les études qui ont précédé notre mémoire ainsi que le prototype développé précédemment [VPMS-87] présentaient de nombreuses lacunes. La plus importante d'entre elles était qu'ils n'envisageaient la présence dans un bureau que d'un nombre beaucoup trop limité de types d'objets et d'opérations primitives. Cette restriction constitue un obstacle important si l'on désire refléter la réalité de la manière la plus précise possible. Conséquence directe: l'outil devient tout à fait inutilisable dans des situations courantes où un analyste peut se trouver face à des objets (exemple: terminaux, télex, machine à écrire, ...) et des opérations qu'il désire pouvoir introduire dans sa simulation alors que ceux-ci ne font pas partie de la liste que le logiciel est apte à simuler. Outre ces lacunes, le prototype présentait de nombreux inconvénients en matière d'interface-utilisateur et ne permettait par exemple pas de créer le décor du bureau de manière à ce qu'il reflète le plus possible la réalité.

Jusqu'à présent, nous n'avons pas encore montré, comme nous l'avions annoncé au chapitre 2, en quoi la programmation orientée objet et ses principes, et en particulier Smalltalk et son architecture, peuvent s'avérer utiles pour résoudre ces problèmes. C'est à quoi nous allons nous attacher dans ce chapitre.

8.1 L'ajout de nouveaux types d'objets

8.1.1 Le mécanisme

Lors de l'exposé de la programmation orientée objet, nous avons écrit qu'une des grandes idées de ce type de programmation était de structurer les systèmes sur base des catégories d'objets manipulés plutôt que sur les fonctions à exécuter. Ces dernières sont, en effet, plus souvent sujettes aux changements que les objets. La méthode de développement consiste alors à déterminer et implémenter les différentes classes d'objets et ensuite à décrire les méthodes associées à chacune des classes. Dans notre cas cependant, même l'ensemble des objets a tendance à se modifier. Nous allons voir que malgré ce phénomène, la programmation orientée objet se montre très utile pour résoudre notre problème.

Dans une programmation orientée objet, ajouter un nouveau type d'objets consiste à ajouter une nouvelle classe implémentant le concept de type abstrait de données. C'est ce dernier concept qui forme sans doute un des points forts de la programmation orientée objet car il donne aux systèmes construits de la sorte, une très grande souplesse et modularité.

Ces deux caractéristiques offrent la possibilité de partir d'un noyau de classes de base et de l'enrichir petit à petit sans altérer les classes de départ. Dans le cas de notre application, on peut considérer que ce noyau de base est composé de l'ensemble des classes définies dans les points précédents (cfr chapitres 6 et 7). Ajouter un nouveau type d'objets revient alors à ajouter une nouvelle classe d'objets au système existant.

Cependant, telle quelle la classe ne peut nous être d'un très grand intérêt. En

effet, nous n'avons pas encore structuré l'état de ses instances sous forme de variables. De même, elle ne possède encore aucune méthode de classe ou d'instance. Cette classe ou ses instances ne peuvent donc pas communiquer avec d'autre classes et/ou instances.

Cela est résolu grâce au mécanisme d'héritage qui donne la possibilité d'organiser différentes classes en arbre dans lequel une classe hérite des variables et méthodes associées à toutes ses super-classes. Dès lors, si pour ajouter un nouveau type d'objets au système, l'analyste crée une nouvelle classe comme sous-classe d'une autre classe, cette nouvelle classe hérite de toutes les propriétés de sa super-classe.

Supposons, par exemple, que l'on dispose des classes A, B, C, D organisées de la manière suivante (figure 8.1 (a)): A est la super-classe de B, B est la super-classe de C et de D. Si l'analyste crée une classe E comme sous-classe de la classe C, sa nouvelle classe hérite alors de toutes les propriétés de ses super-classes C, B et A (figure 8.1 (b)).

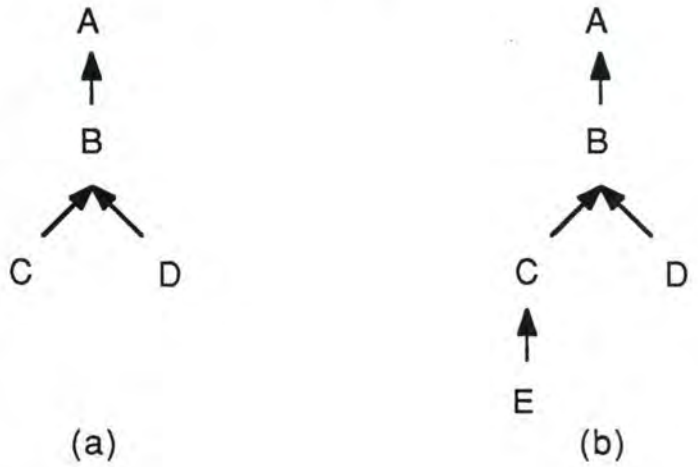


figure 8.1: Un exemple d'ajout de classe

8.1.2 Avantages de notre architecture

Ce sont les différents principes exposés au point 8.1.1 qui ont été appliqués lors de la création de notre logiciel. De plus, l'emploi intensif des principes de généralisation et d'abstraction pour déterminer la hiérarchie des classes du noyau de base nous a permis de créer une architecture rendant le problème d'ajout de nouveaux types d'objets parfaitement soluble.

8.1.2.1 La hiérarchie des classes de modèles

L'utilisation des principes de généralisation et d'abstraction lors de la définition des classes de modèles a donné lieu à une architecture dans laquelle les classes inférieures (Message, Formulaire, ...) ne possèdent que très peu d'informations propres. Par informations, nous entendons des variables et méthodes particulières à leur modélisation. Nous pouvons également désormais justifier la

présence de variables dans certaines classes alors que leur présence s'expliquerait apparamment mieux dans une des sous-classes. Le fait d'avoir remonté dans les super-classes toutes les variables d'instance ainsi que les méthodes de création d'environnement, d'opérations primitives et de structures d'enchaînement, permet non seulement d'éviter des redondances inutiles mais aussi de faciliter l'ajout de nouvelles classes d'objets. En effet, les nouvelles classes peuvent hériter automatiquement de toutes les propriétés des classes de base Composant, Information, Rangement, Travail ou Interface. Grâce à cet héritage, il n'est pas nécessaire de passer par une nouvelle phase de déclaration de variables et de codage des méthodes nécessaires à leur comportement.

Ainsi, par exemple, supposons que l'on désire ajouter au système une classe Enveloppe et que l'on désire modéliser ses instances comme des objets de rangement, il suffit de créer la nouvelle classe comme sous-classe de la classe Rangement au même titre que le sont les classes Armoire, Tiroir, Etagere, Boîte et Farde. On aurait alors la hiérarchie exposée à la figure 8.2.

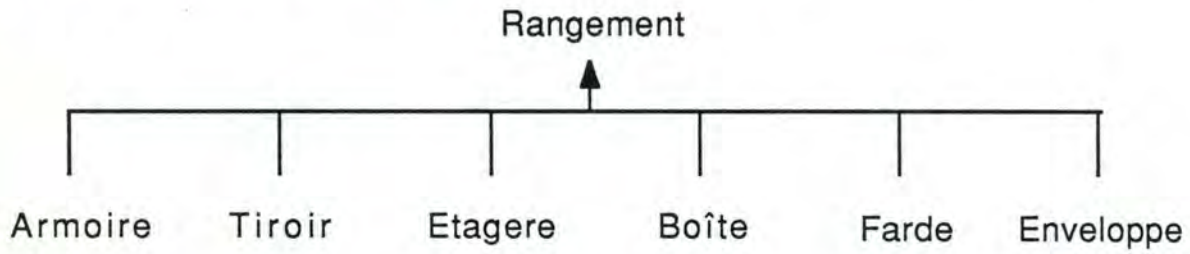


figure 8.2: Ajout de la classe Enveloppe

Dans cet exemple, la structure de l'état d'une instance de la classe Enveloppe est la même que tout objet de rangement puisqu'elle hérite des variables d'instance et de classe de toutes ses super-classes. Son comportement est également identique à tout objet de rangement puisqu'elle possède les méthodes définies dans ses super-classes.

Puisque toutes les variables et méthodes nécessaires aux objets de rangement se trouvent dans les classes Rangement et Composant, Enveloppe peut être utilisée comme l'est toute sous-classe de Rangement. On peut par exemple créer une instance de cette classe et l'associer à un bureau puisque le message de création est hérité de la classe Rangement. On peut effectuer des associations de composition entre une instance de la classe Enveloppe et un autre objet de rangement, le message de création d'association étant également hérité de la classe Rangement et on peut ranger des objets informationnels dans une enveloppe puisqu'elle hérite des structures de données propres à tout objet de Rangement.

Au niveau des modèles, on peut donc considérer que le problème d'ajout d'objets est résolu. Il reste à examiner s'il l'est tout autant pour ce qui concerne les vues et les modèles.

8.1.2.2 La hiérarchie des classes de vues et de contrôleurs

La façon dont nous avons géré l'affichage des objets et la communication

entre ceux-ci et les utilisateurs permet d'utiliser un nouveau type d'objets de la même manière que ceux déjà définis.

En effet, au chapitre 7, nous avons vu que tout objet de bureau, instance d'une sous-classe de Composant, qui était affiché à l'écran était associé à une vue, instance de la classe ComposantView et à un contrôleur, instance de la classe ComposantController. Ces deux classes regroupent toutes les structures de données et méthodes nécessaires à l'affichage des objets et à leur interaction avec les utilisateurs. Quant à la classe Composant, c'est elle qui fournit les méthodes utilisées par les objets pour coopérer avec leurs vues et contrôleurs; toutes ses sous-classes, les nouvelles en particulier, en héritent donc. Ceci implique qu'une instance d'une nouvelle classe sera automatiquement associée à une instance de la classe ComposantView et de la classe ComposantController, sa présentation sera donc gérée de la même façon que tout autre objet. Ces relations entre les différentes classes sont illustrées à la figure 8.3.

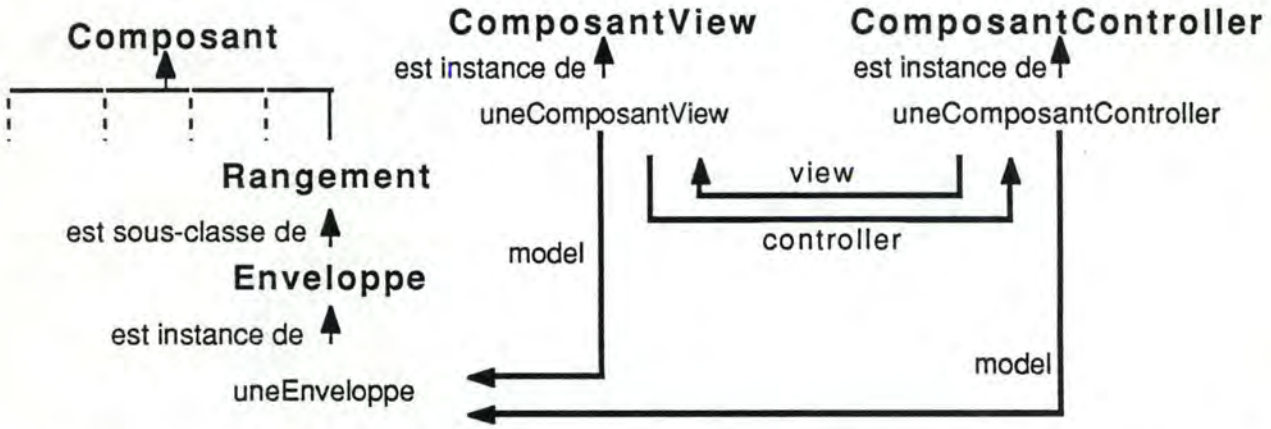


figure 8.3: Les relations existants entre les classes Composant, ComposantView et ComposantController

8.1.3 Le travail de l'analyste

Comme nous venons de le voir, l'analyste qui souhaite ajouter un nouveau type d'objets doit insérer une nouvelle classe dans la hiérarchie du système Smalltalk.

Pour réaliser cela, il peut utiliser les vues System Browser (point 3.4.5) qui font partie de l'environnement de programmation. Toutefois, l'utilisation de ces vues requiert une certaine connaissance de Smalltalk. Or un bon logiciel doit cacher le plus possible les détails de son implémentation. Dès ce moment, il faut faire en sorte que lors de l'ajoût, l'analyste ait l'impression de créer un nouveau type d'objets et non pas une classe Smalltalk.

C'est pourquoi , notre idée est de fournir un outil du même type que les "System Browser" (point 3.4.5) du système Smalltalk mais qui serait plus facile d'emploi. Cet outil permettrait de parcourir la hiérarchie des types d'objets de bureau et d'en rajouter de nouveaux tout en cachant à l'analyste les notions propres au système Smalltalk. On offrirait aussi la possibilité de supprimer et de renommer chaque type d'objets, ainsi

que de modifier le dessin qui leur est associé.

Afin de créer cet outil, nous avons défini la classe **InfoBrowser**. Cette nouvelle classe est sous-classe de la classe Browser (figure 8.4) qui détermine les caractéristiques propres aux "System Browser" de l'environnement Smalltalk et dont l'architecture a été exposée au point 4.3.2. Notre classe hérite donc des propriétés de la classe Browser et en constitue une spécialisation en y ajoutant ses propres variables et méthodes.

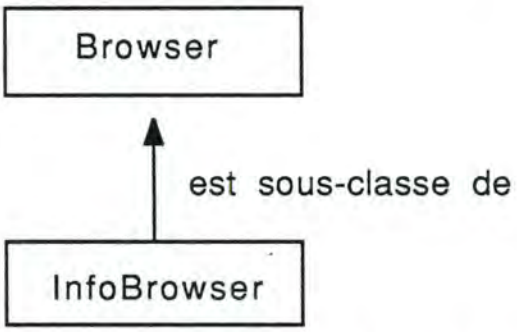


figure 8.4: La classe InfoBrowser et sa super-classe

La conception de notre outil de création de nouveaux types d'objets se base également sur l'architecture Modèle-Vue-Contrôleur (point 4.2). Cependant, nous n'avons pas créé de classes spécifiques pour les vues et contrôleurs associés aux instances de la classe InfoBrowser. En effet, grâce au mécanisme d'héritage, nous pouvons utiliser automatiquement les classes nécessaires et rattachées à la classe Browser. Tout au plus, avons-nous dû rajouter ou modifier certains messages dans notre nouvelle classe. Ceci est donc une illustration pertinente du phénomène de réutilisabilité (point 3.1.2) présent en programmation orientée objet.

Lors de l'ajoût d'un nouveau type d'objets, c'est-à-dire d'une classe, l'instance de la classe InfoBrowser se charge de créer la définition Smalltalk de la nouvelle classe. Elle réalise cela à partir du nom donné par l'utilisateur et de la super-classe choisie. Elle compile cette définition et ajoute la classe à la hiérarchie existante. La classe InfoBrowser contient également des méthodes servant à renommer, supprimer un type d'objets et à changer le dessin qui lui est associé à l'aide d'un éditeur graphique.

Cette dernière fonctionnalité est très utile car elle donne à l'analyste la possibilité de représenter les objets de la manière la plus proche possible de la réalité du bureau qu'il analyse. En outre, elle s'avère également très intéressante car lors de l'ajout d'un nouveau type d'objets, on associe à ce dernier une forme par défaut, ce qui implique que tous les nouveaux objets créés auront la même apparence à l'écran.

Le mécanisme de modification de la forme se réalise de la manière suivante. Pour obtenir le dessin associé à un objet de bureau, le principe est de lui envoyer le message **form**. La méthode déclenchée par la réception de ce message renvoie la forme associée à l'objet. Puisque par défaut, tous les objets de bureau ont la même forme, le message form a été implémenté dans la classe Composant. Toutes ses sous-classes en héritent donc. Modifier la forme d'un type d'objets provoque l'ajout

automatique, dans sa classe, d'un message de sélecteur qui renvoie la forme adéquate. Les figures 8.6 (a) et 8.6 (b) illustrent la modification de la forme associée aux instances de la classe Farde.

Avant la modification (figure 8.6 (a)), la classe Farde ne possède pas de message form propre dans son protocole, elle l'hérite de la classe Composant. Après la modification (figure 8.6 (b)), le message form y a été ajouté. Lors de l'envoi de ce message à une instance de cette classe, la recherche de la méthode dans la hiérarchie s'arrêtera lorsqu'on aura rencontré le premier message form. On aura donc bien la forme voulue. Ceci constitue un exemple de surcharge évoquée au chapitre 3.

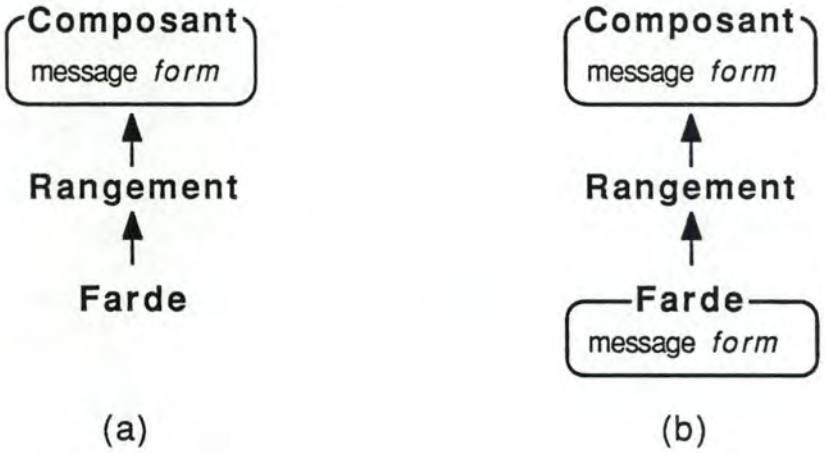


figure 8.6: Modification de la forme associée à la classe Farde

La figure 8.5 montre notre outil d'ajout de nouveaux types d'objets.

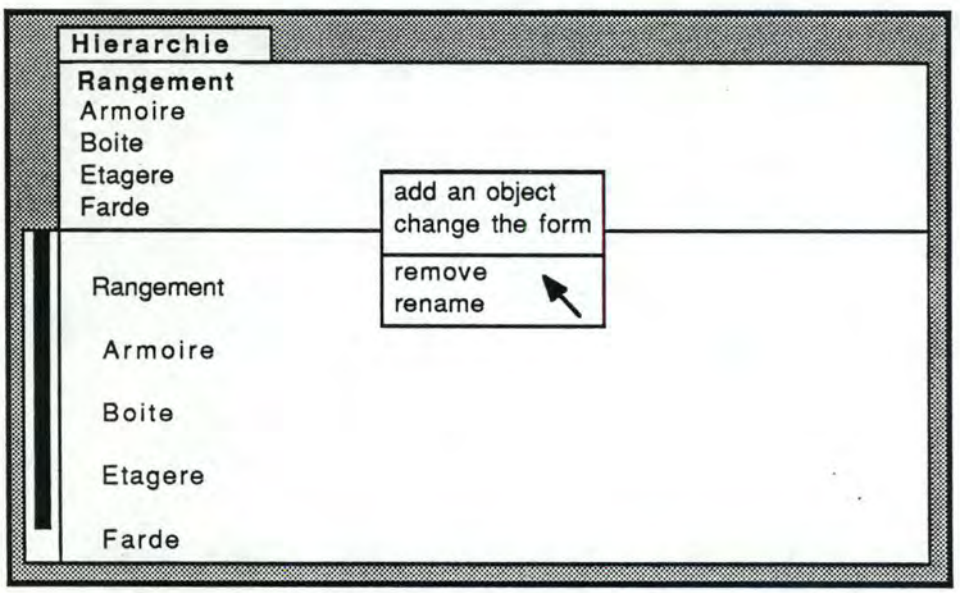


figure 8.5: L'outil d'ajout de nouveaux types d'objets

8.2 L'ajout de nouvelles opérations primitives

8.2.1 Le mécanisme

Au chapitre 2, nous avons dit qu'une tâche de bureau pouvait se décomposer en opérations primitives. Ces opérations sont applicables à des objets informationnels de bureau, et à priori quel que soit leur type. C'est pourquoi, elles ont été implémentées sous forme de messages, les méthodes qui les décrivent appartenant à la classe information, toutes les sous-classes de cette classe peuvent ainsi en hériter. Ces messages font partie du protocole de classe de la classe Information et doivent être envoyés à cette classe ou à une de ses sous-classes.

En conséquence rajouter une opération primitive consiste simplement à ajouter un nouveau message à l'interface de la classe Information. Ainsi, toutes ses sous-classes peuvent en hériter et la nouvelle opération peut leur être appliquée.

Tel est le principe de base de notre implémentation. Cependant, il faut remarquer qu'à chaque opération primitive correspondent des préconditions (point 4.7) qui doivent être respectées avant son exécution. Dès lors, à chacune des opérations primitives correspondent un second message et sa méthode. Cette méthode est chargée de vérifier si les préconditions sont vérifiées avant l'exécution de l'opération, et fait également partie du protocole de classe de la classe Information.

Exemple:

Soit le message

communiquer: unNom **numCopie**: unNumCopie **type**: unType
bureau: unBureau

qui permet de communiquer, par la boîte OUT, un objet informationnel dont la classe est la classe réceptrice, le nom est unNom, le numéro de copie est unNumCopie et le type est unType.

A ce message correspond un autre message

precondcommuniquer: unNom **numCopie**: unNumCopie **type**: unType
bureau: unBureau

qui, en fonction des arguments qui lui sont transmis, contrôle si les préconditions sont respectées. Dans notre exemple, il vérifie si la boîte OUT et l'objet informationnel spécifié existent dans le bureau unBureau. Si ce n'est pas le cas, la simulation est arrêtée et l'analyste est averti.

Toute méthode décrivant une opération primitive, débute par l'envoi du message de vérification des préconditions à la classe réceptrice du message. Dans une méthode, le récepteur du message est contenu dans la pesudo-variable **self**. Ainsi dans le cas de la communication par la boîte OUT, la première expression de la méthode est:

self **precondcommuniquer:** unNom **numCopie:** unNumCopie **type:** unType **bureau:** unBureau.

8.2.2 Le travail de l'analyste

De la même façon que l'ajoût de nouveaux types d'objets, l'ajout d'opérations primitives dans le système peut se faire grâce aux vues System Browser du système Smalltalk. Cependant, nous proposons également un outil plus aisé d'emploi. Cet outil constitue un "browser" simplifié qui permet de parcourir la liste des méthodes représentant les opérations primitives et les préconditions, de modifier des méthodes, d'en ajouter ou d'en supprimer.

A cet effet, nous avons créé la classe **OperationBrowser** comme sous-classe de la classe Browser (figure 8.7).

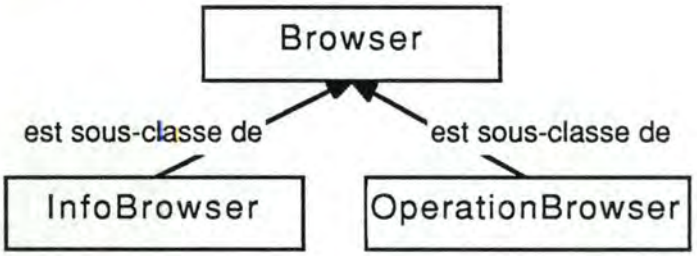


figure 8.7: La classe OperationBrowser et sa super-classe

Tout comme la classe InfoBrowser, notre nouvelle classe constitue une spécialisation de sa super-classe en ajoutant certaines variables et méthodes nécessaires à son fonctionnement. Elle utilise également de manière indirecte les mêmes classes que la classe Browser en ce qui concerne la gestion des vues et contrôleurs. La figure 8.8 montre notre outil d'ajout d'opérations primitives.

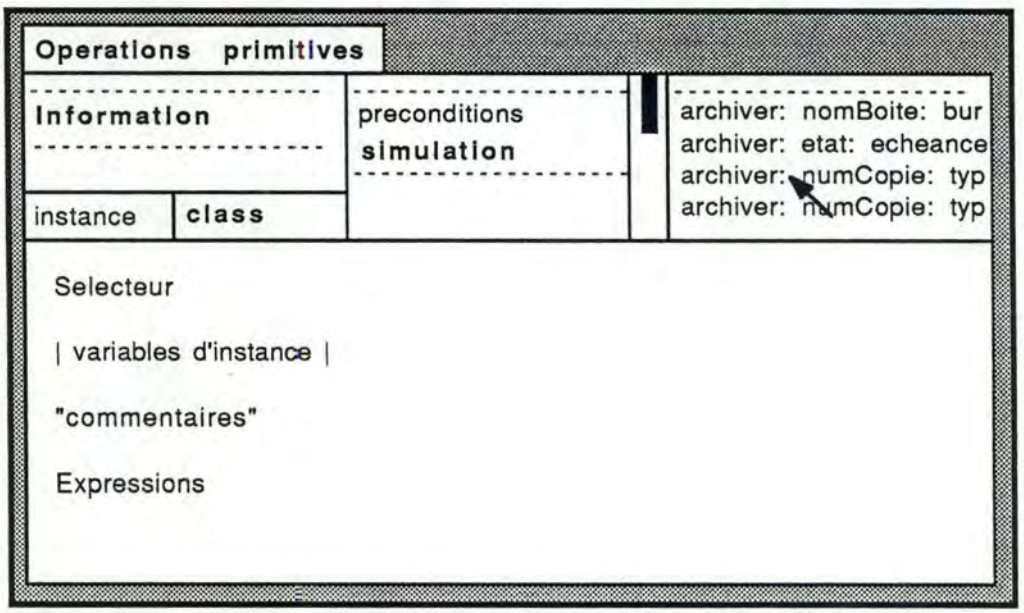


figure 8.8: L'outil d'ajout de nouvelles opérations primitives

8.3 Utilisation des vues et contrôleurs

L'architecture Modèle-Vue-Contrôleur telle que nous l'avons exposée au chapitre 4 et telle que nous l'avons implémentée dans notre logiciel offre de nombreuses possibilités en matière de graphisme et d'interface-utilisateur.

Dans ce point 8.3, nous allons rapidement décrire quelques exemples de ce que nous avons réalisé à ce niveau. Cette réalisation constitue, par ailleurs, une amélioration notable par rapport au prototype conçu par [VPMS-87], tant au niveau de la définition de l'environnement de bureau qu'au niveau de la simulation d'une tâche.

En effet, ce premier prototype présentait quelques défauts qu'il nous a semblé nécessaire de supprimer (point 2.5.2). Entre autres, il n'offrait pas à son utilisateur, la possibilité de déplacer à l'écran les objets du bureau ou d'identifier, par leur nom, ces mêmes objets.

Cette possibilité de modification des vues et contrôleurs est facilitée grâce à la manière dont nous avons fait usage de l'architecture M.V.C. dans notre travail. En effet, le fait que tout objet de bureau à présenter à l'écran soit associé à une instance de la classe ComposantView et à une instance de la classe ComposantController (point 5.2) a pour conséquence que chaque objet est présenté de la même façon et offre des possibilités identiques en ce qui concerne la communication avec les utilisateurs. Dès lors, toute modification apportée à une de ces deux classes est ressentie de façon similaire par tous les objets de bureau.

Afin de créer un interface-utilisateur agréable, nous avons utilisé les messages et variables décrites dans la classe MouseMenuController (point 5.2); de la sorte, un Pop-Up menu particulier est associé à chaque objet présenté à l'écran. Ce Pop-Up menu est accessible au moyen de la souris et possède trois options:

- le *déplacement* de l'objet dans la fenêtre du bureau
- l'*identification* de l'objet
- l'*obtention d'un répertoire* reprenant les différents objets se trouvant dans ou sur un objet

Cette liste d'options n'est toutefois pas limitative. D'autres possibilités peuvent être ajoutées au menu, simplement par modification des messages appropriés de la classe ComposantController.

8.3.1 Quelques améliorations de la définition de l'environnement de bureau

Afin que la communication entre l'analyste et l'utilisateur soit réussie, il est nécessaire que ce qui sera montré à l'écran soit une représentation la plus fidèle possible de l'environnement dans lequel l'utilisateur est habitué à travailler.

Pour éviter de toujours présenter à l'écran une image figée d'un bureau-type, comme c'était le cas dans [VPMS-87], la possibilité est donnée à l'analyste de ne créer que les objets présents dans le bureau qu'il est chargé d'automatiser. Il a également la possibilité de déplacer ces objets pour qu'ils occupent, dans la représentation du bureau, une place identique à celle qu'ils occupent dans la réalité.

8.3.1.1 La création des objets du bureau

La création des objets se fait à l'aide de messages Smalltalk (point 6.6.2). Ces messages permettent de créer les objets strictement nécessaires et de leur donner un nom particulier. L'envoi de ces messages aux classes adéquates permet d'afficher exclusivement les objets définis. Pour des raisons de graphisme et d'esthétisme, et comme c'est également le cas dans la réalité, nous avons décidé de ne représenter que les objets qui ne sont ni rangés, ni classés dans un autre objet ou qui ne composent aucun autre objet.

Comme nous l'avons vu plus haut, à partir de chaque objet présenté, l'utilisateur peut accéder à un Pop-Up menu contenant trois options. Chacune de ces options doit l'aider dans la description du décor dans lequel vont se dérouler les simulations de tâches. Nous allons examiner maintenant chacune de ses options.

a) Le déplacement d'un objet

Grâce à cette option, les objets peuvent être placés là où l'utilisateur le désire, selon la disposition de son propre bureau. En effet, le contrôleur associé à la vue de l'objet permet de le déplacer à l'aide de la souris, dans les limites de la fenêtre représentant le bureau (figures 8.9 et 8.10). Cependant, il subsiste une contrainte d'utilisation. Lorsque l'on veut bouger un objet, il est imposé que tous les objets qui le chevauchent doivent être déplacés auparavant, ceci simplement pour des raisons d'implémentation graphique..

Exemple: Les messages suivants ont pour but de créer des objets dans un bureau de nom secrétariat:

```
b <- Bureau creer: #secrétariat.
TableTravail creerDans: b.
Photocopieuse creerDans: b.
Poubelle creerDans: b.
Armoire creer: #armoire1 bureau: b.
Etagère creer: #étagère1 bureau: b.
```

L'exécution de l'expression b **ouvrirDécor** a pour effet d'afficher à l'écran tous les objets définis par les messages précédents (figure 8.9).

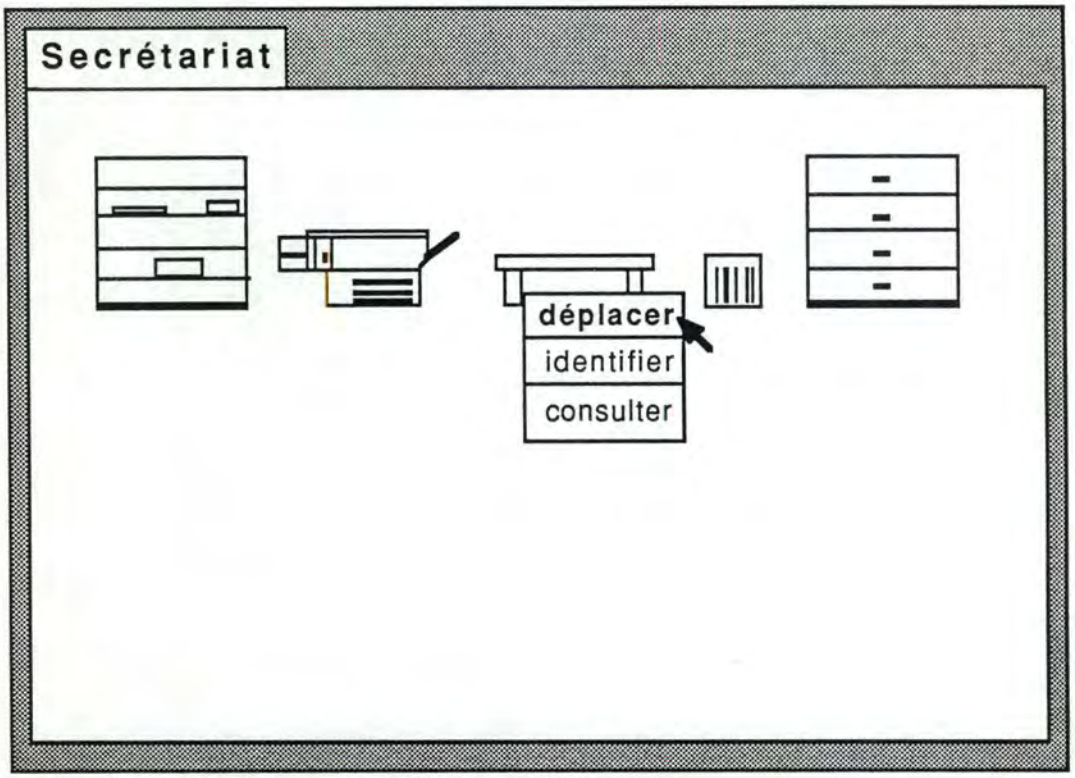


figure 8.9: Les objets du bureau avant déplacement

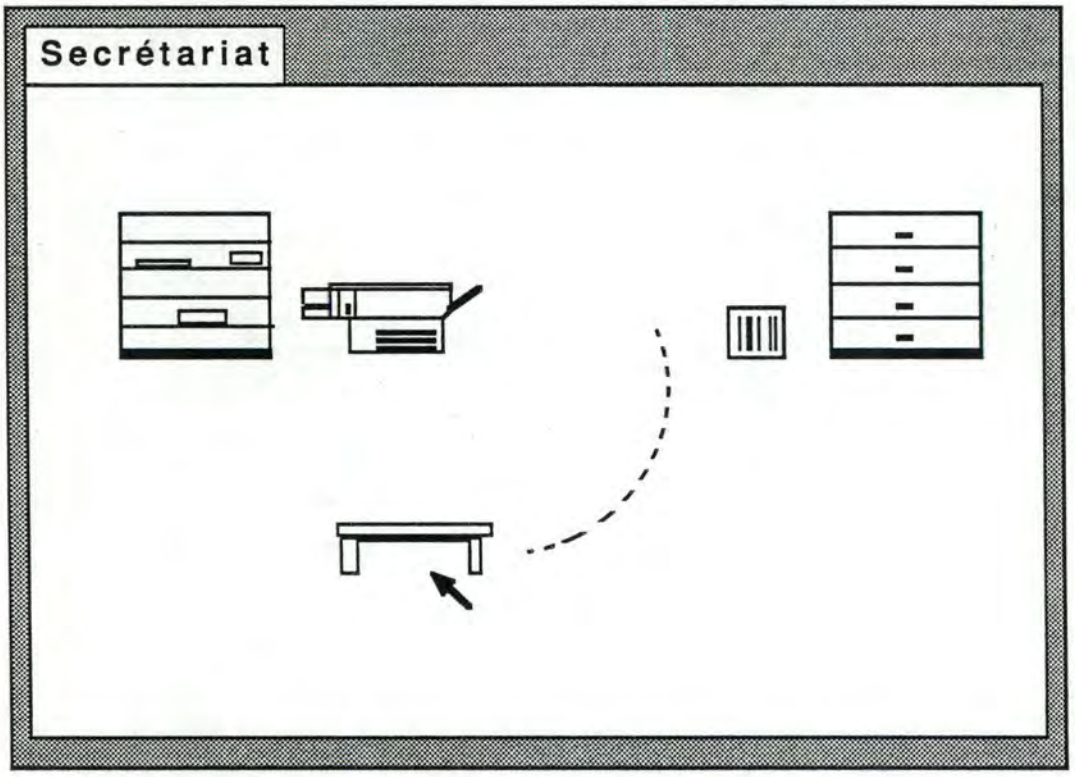


figure 8.10: Les objets du bureau après déplacement

Lors du déplacement d'un objet, la coopération entre les trois composants modèle, vue et contrôleur s'effectue de la manière suivante: dès que l'utilisateur a spécifié au contrôleur qu'il désire déplacer l'objet auquel il est associé, ce contrôleur

signifie à sa vue qu'à partir de ce moment elle doit suivre les mouvements de la souris jusqu'à ce qu'un de ses boutons soit pressé. Lorsque le bouton a été pressé, la vue donne l'ordre à son modèle d'enregistrer la nouvelle position qu'il occupe dans la fenêtre du bureau.

b) L'identification d'un objet du bureau

Une identification la plus précise possible de chaque objet présent dans le bureau est nécessaire, tant en cours de simulation d'une tâche qu'en cours de définition de l'environnement du bureau. Cette identification permet de repérer un objet et de distinguer deux objets d'apparence identique mais de nom différents. En effet, chaque objet appartenant à la même classe est représenté à l'écran par une même forme (point 8.1.3).

L' identification d'un objet se fait au moyen d'une étiquette et grâce au Pop-Up menu. Le nom de l'objet s'affiche jusqu'à ce qu'un bouton de la souris soit pressé (figures 8.11 et 8.12).

Exemple:

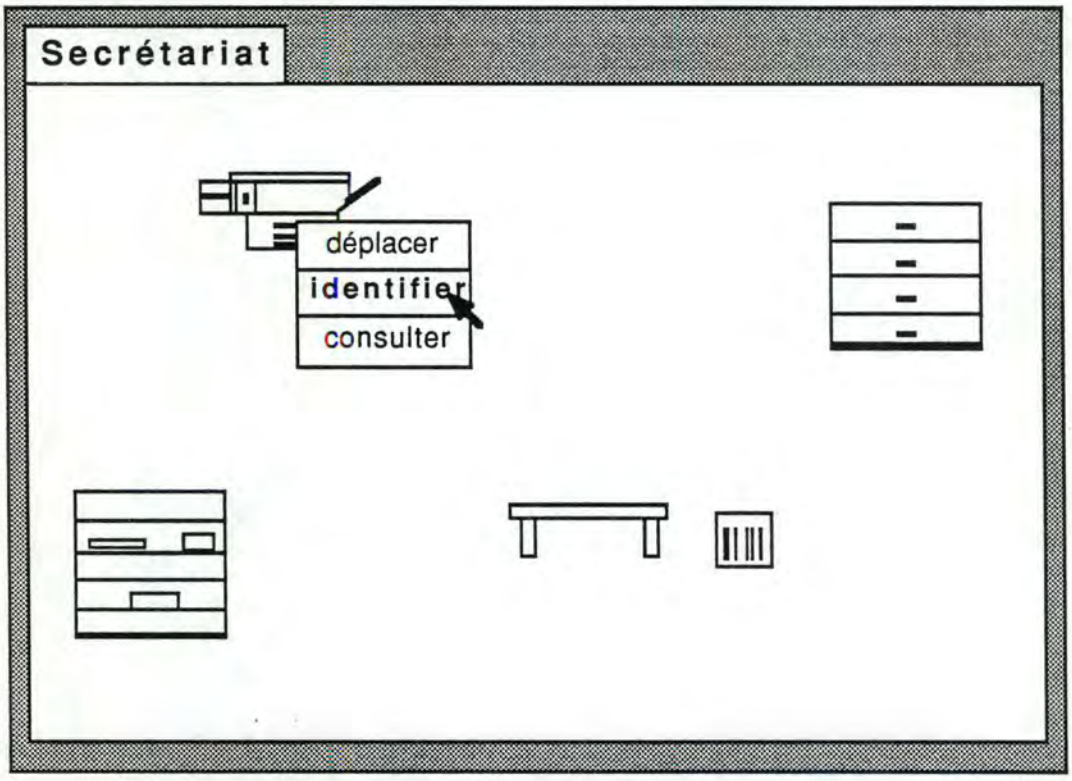


figure 8.11: Les objets du bureau avant identification

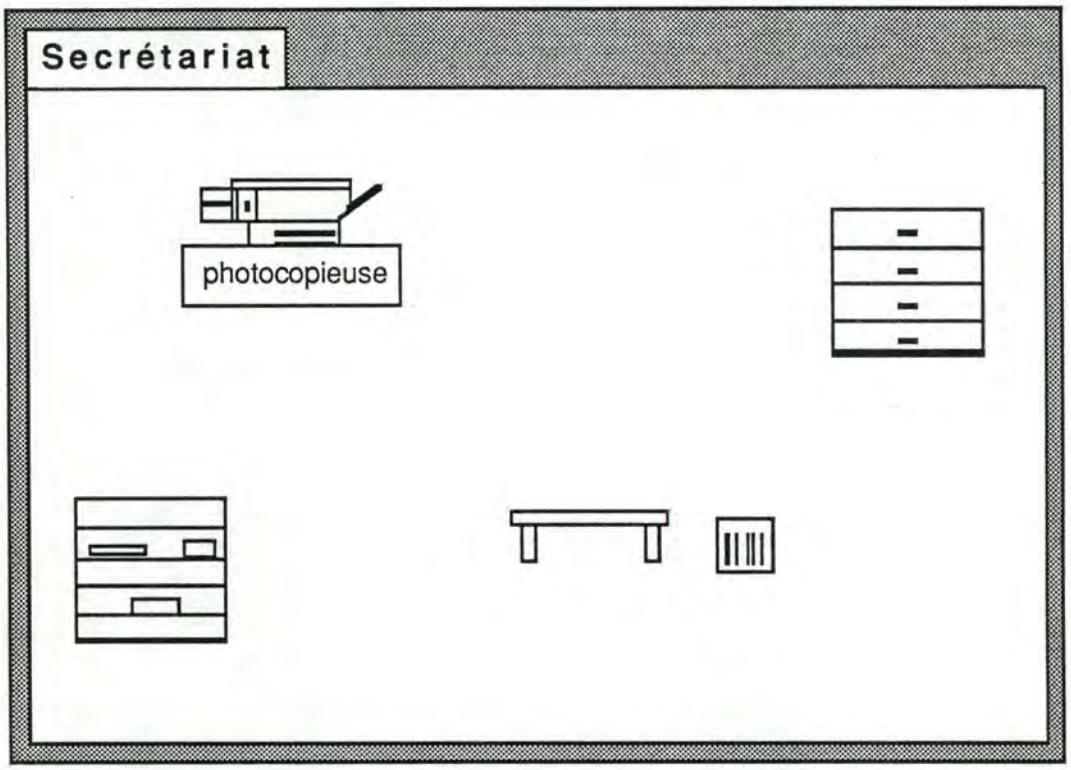


figure 8.12: Les objets du bureau après identification

c) L'obtention du répertoire d'un objet

Cette option du pop-up menu permet d'afficher un répertoire hiérarchique de l'objet désigné (figures 6.5 et 6.6).

Exemple:

L'armoire de nom *armoire* contient le tiroir de nom *tiroir*. Celui-ci contient la farde de nom *farde* contenant le dossier de nom *dossier*. *Dossier* renferme le message de nom *message* et de numéro de copie 0 et le formulaire de nom *formulaire* et de numéro de copie 1. Le répertoire se présente comme ceci:

Répertoire de *armoire*
tiroir: *tiroir*
farde: *farde*
dossier: *dossier**
message: *message* (0)
formulaire: *formulaire* (1)

L'astérisque indique que le dossier ne se trouve pas physiquement dans la farde au moment de la consultation. De plus, comme nous avons décidé de ne représenter que les objets qui ne sont ni rangés, ni classés dans un autre objet et qui ne composent aucun autre objet, la farde ne sera pas représentée à l'écran, mais le répertoire permettra de savoir qu'il existe effectivement une farde dans le tiroir.

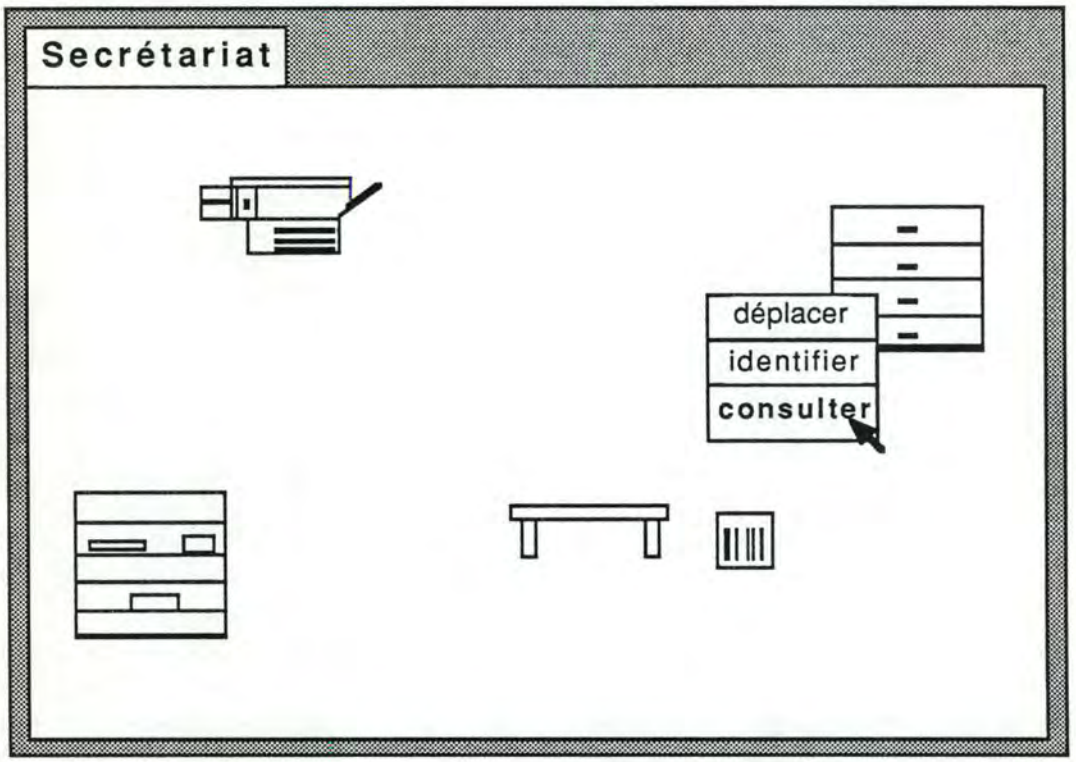


figure 8.13: Les objets du bureau avant consultation du répertoire

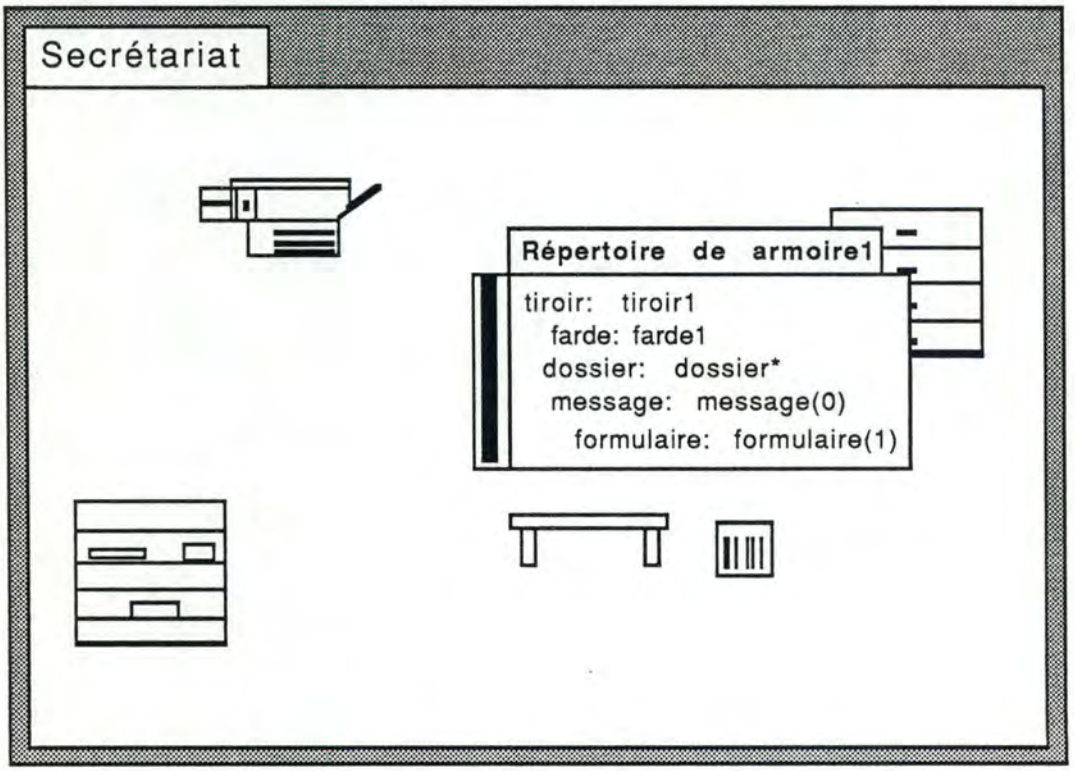


figure 8.14: Les objets du bureau après consultation du répertoire

8.3.2 Améliorations de la simulation d'une tâche

Dans le prototype (point 2.5), un des gros inconvénients qui apparaissaient lors de la simulation d'une tâche, venait de ce qu'il était difficile de se rendre compte de ce qui se passe réellement à l'écran en raison du manque d'une identification correcte des objets se trouvant ou se déplaçant dans le bureau. Une amélioration a été mise au point par affichage du nom de tout objet jouant un rôle dans la simulation de la tâche.

Avant tout déplacement d'un objet informationnel au cours de la simulation d'une tâche, notre logiciel affiche le nom de l'objet de rangement, de travail ou d'interface dans lequel se trouve l'objet informationnel en question. Ensuite, apparait l'objet informationnel lui-même ainsi que son nom, et enfin, le nom de l'objet de destination.

Exemple:

Lors de la simulation d'une tâche, le message de nom *message* se trouvant dans l'armoire de nom *armoire1* doit se déplacer vers l'étagère de nom *étagère* pour y être rangé. Avant le déplacement du message, la représentation du bureau se présente comme suit (figure 8.15):

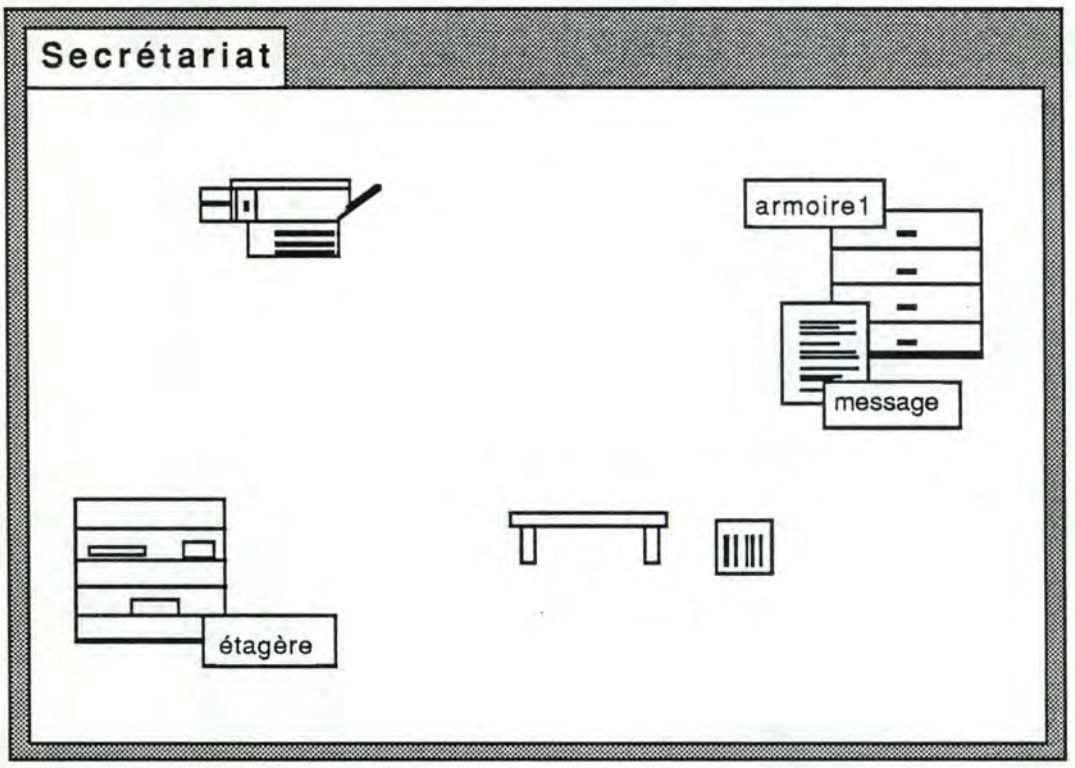


figure 8.15: L'dentification d'un objet avant son déplacement dans la fenêtre du bureau

CHAPITRE 9

CONCLUSION ET PERSPECTIVES D'AVENIR

CHAPITRE 9: CONCLUSION ET PERSPECTIVES D'AVENIR

Dans les chapitres précédents, nous avons exposé l'implémentation de notre outil et les améliorations apportées aux travaux antérieurs. Ce chapitre est consacré à une brêve critique de l'environnement de programmation Smalltalk-80, à une synthèse de notre travail et à quelques propositions pour le développement futur du projet.

9.1 Critique de l'environnement de programmation Smalltalk-80

L'utilisation de Smalltalk-80, ainsi que de nombreuses lectures, nous ont permis de percevoir les avantages de ce système mais aussi ses principaux défauts.

Smalltalk-80 étant un langage orienté objet, il est normal qu'il présente tous les avantages de ce type de programmation (extensibilité, réutilisabilité, ...). Cependant, vu l'application extrêmement poussée des principes décrits précédemment, ses points forts sont encore renforcés. Ainsi, les notions d'objet et d'information cachée permettent de réduire la complexité des systèmes en minimisant les interdépendances entre les composants d'un programme et en permettant la réalisation de changements incrémentaux de ce même programme. Ceci couplé avec un environnement de travail très développé, fait de Smalltalk-80 un excellent outil, comme nous le verrons au point suivant.

Toutefois, on peut émettre quelques critiques. La première est qu'il n'existe aucune possibilité de typage des données. Cela signifie qu'aucun contrôle n'est effectué à la compilation, reportant alors la détection d'éventuelles erreurs lors de l'exécution des programmes. Ensuite, nous pourrions également dire que Smalltalk est un environnement très fermé qui ne s'exécute actuellement que sur certaines machines dédicacées. Un programme écrit dans ce langage ne pourrait pas, par exemple, être installé sur une autre machine et/ou communiquer avec un autre programme rédigé dans un langage plus conventionnel tel que C, PASCAL, ... C'est pourquoi, ses applications principales sont la construction de prototypes de systèmes qui seront réécrits par la suite à l'aide d'autres outils [DIED-87].

Nous retiendrons également que la multiplicité et la richesse des classes de base rend l'apprentissage et la maîtrise de Smalltalk assez difficile. En effet, du fait de leur nombre et de la complexité de l'arbre d'héritage, il n'est pas toujours facile de trouver la classe qui réponde le mieux aux besoins du programmeur. Ceci est accentué par le fait que le système pêche par un manque de documentation valable notamment en ce qui concerne les classes d'objets permettant la création d'outils graphiques et d'interface-utilisateurs (multi-fenêtres, Pop-Up menus, ...) Dès lors, la seule manière d'apprendre à utiliser le système avec un minimum d'efficacité reste l'expérience.

Comme nous l'avons écrit, tout est un objet (point 3.4.2.1) en Smalltalk-80, même l'Operating System. Ceci signifie qu'en Smalltalk, il n'existe pas d'Operating System protégé et donc inchangeable. Le code source du sytème et son environnement de programmation tout entier peuvent être directement modifiés de n'importe quelle manière en changeant simplement quelques lignes de codes.

Le dernier problème engendré par l'utilisation de Smalltalk-80 est la difficulté d'introduire un contrôle entre les différentes personnes travaillant ensemble sur le système en organisation coordonnée. En effet, la capacité que possède n'importe quel programmeur de changer tout ce qu'il désire ne paraît pas constituer un avantage dans ce genre de travail.

9.2 Les apports de notre mémoire

Malgré ces inconvénients, l'environnement Smalltalk reste néanmoins un outil très puissant de développement de programmes qui nous a permis de développer une deuxième version du prototype réalisé précédemment [VPMS-87], en supprimant ses défauts majeurs. Il est vrai que le langage C et le logiciel MS-Window qui avaient servi lors de son implémentation n'étaient pas les mieux adaptés à ce genre de travail.

Ainsi, grâce aux concepts de types abstraits de données et d'héritage propres aux langages orientés objets, un système d'ajoût d'objets et d'opérations primitives a pu être mis au point. Cela signifie qu'à tout moment, un analyste peut désormais définir les objets et les opérations primitives dont il a besoin pour l'analyse d'un bureau.

Par l'utilisation de l'architecture M.V.C. et des outils graphiques, les définitions d'environnement et les simulations de tâches ont pu être fortement améliorées. L'analyste a, par exemple, la possibilité de définir correctement le décor de son bureau à l'écran par déplacement des objets à l'aide de la souris et peut obtenir des informations précises concernant les objets présentés. Ceci permet de représenter plus fidèlement la réalité et d'améliorer le dialogue entre l'analyste et l'utilisateur.

En outre, de par sa très grande modularité, notre programme peut facilement être adapté aux changements éventuels des spécifications, et des améliorations notables peuvent encore lui être apportées sans altérer le noyau existant.

Notre expérience de Smalltalk nous a laissé entrevoir quelques-unes de ces améliorations. Chacune d'elles constitue le point de départ d'un développement futur du projet. La plupart des améliorations possibles fait l'objet, dans les points suivants, d'une réflexion de notre part et est, à notre avis, réalisable. Cependant, par manque de temps, nous n'avons pas pu les implémenter.

9.3 Les perspectives d'avenir

9.3.1 Création d'un outil de modification des contraintes liées aux associations entre les objets de bureau

Comme nous l'avons vu (point 6.5), afin de définir son environnement de bureau, l'analyste doit créer un ensemble d'objets et d'associations entre ces objets. Il fait cela par des envois de messages à des classes (point 6.5.2).

Lors de l'établissement de ces associations, l'analyste doit respecter certaines contraintes. Ainsi, on ne peut pas classer une pile dans un autre objet informationnel structurant, ou encore une armoire ne peut être composée que de tiroirs. Le respect de toutes ces contraintes est vérifiée par le système.

Le principe de vérification est le suivant: à chaque message de création d'association correspond un message (point 8.2.1) possèdant les mêmes arguments et qui permet de vérifier si les contraintes liées à cette association sont respectées ou pas (point 6.5.3). Ces deux messages font partie du protocole de la même classe. L'envoi d'un message de création d'association à une classe déclenche l'exécution d'une méthode dont la première expression consiste à envoyer le message de vérification des contraintes à la classe réceptrice du message. Ce nouveau message déclenche l'exécution d'une seconde méthode. Cette dernière contrôle, en fonction des arguments qui lui sont transmis, si toutes les contraintes sont bien respectées. Si tel est le cas, elle repasse le contrôle à la première méthode qui se charge de créer l'association proprement dite. Si les préconditions ne sont pas vérifiées, la méthode arrête l'exécution du programme et affiche un message à l'écran invitant l'analyste à corriger la définition de l'environnement.

Afin d'illustrer cette manière de travailler, supposons que l'on désire construire une association de composition entre une étagère de nom #etagere et une boîte de nom #boite dans le bureau b. Pour ce faire, il faut exécuter l'expression suivante:

```
Boite associationCompEntre: #boite
                            et: #Etagere
                            nomComposé: #etagere
                            bureau: b.
```

Ceci a pour effet de déclencher l'exécution de la méthode dont le texte apparait à la figure 9.1 et qui est déclarée dans le protocole de classe de la classe Rangement.

```
associationCompEntre: unNom1 et: UneClasseComposé
nomComposé: unNom2 bureau: un bureau

|objet1 objet2 precond|

self precondEntre: unNom1 et: uneClasseComposé
     nomComposé: unNom2 bureau: unBureau.
objet1 <- self instance: unNom1.
objet2 <- (Smalltalk at: uneClasseComposé) instance: unNom2.
objet1 estContenuDans: objet2.
objet2 majDictionnaire: objet1.
```

figure 9.1: La méthode décrivant le message de création d'association de composition

Dans cette méthode, la pseudo-variable self désigne le récepteur du message qui a déclenché l'exécution de la méthode. La première expression consiste dès lors à envoyer le message *precondEntre: unNom1 et: uneClasseComposé nomComposé: unNom2 bureau: unBureau* à la classe réceptrice du message de création d'association. Le nouveau message déclenche, quant à lui, l'exécution d'une méthode qui se charge de contrôler si l'association peut être créée entre les deux objets

spécifiés par les arguments. Si tel est le cas, l'exécution de la méthode se termine et la méthode de la figure 9.1 se charge de créer l'association. Sinon le programme s'arrête et l'analyste est averti qu'on ne peut réaliser l'association.

Le principe est le même pour chaque type d'association. Le message de vérification des contraintes relatives aux messages de composition est défini dans la classe Rangement. Quant aux messages utilisés dans le cas des autres associations, ils font partie du protocole de la classe Information.

Au chapitre 8, nous avons exposé le fonctionnement de notre outil d'ajoût de nouveaux types d'objets. Bien que ce système d'ajoût fonctionne correctement, il présente nénmoins un inconvénient relativement important. En effet, une instance de la nouvelle classe va hériter des propriétés propres à toutes les instances de ses super-classes. Or, si on désire spécifier des contraintes particulières aux nouveaux objets, elles devront être ajoutées par l'analyste, en Smalltalk bien sûr, dans une méthode vérifiant les préconditions de classement, de rangement, de composition, de travail ou d'interface selon les cas.

Exemple:

> Supposons que l'on ajoute un nouveau type d'objets de rangement appelé Enveloppe. Si l'on désire spécifier que l'on ne peut ranger des objets informationnel structurants dans une enveloppe, il faut alors modifier, en Smalltalk, la méthode définissant le message dont le sélecteur est precondAssRgt: numCopie: unType: et: nom: bureau:. Cette méthode est chargée de vérifier si les préconditions sont respectées avant de créer une association de rangement entre un objet informationnel et un objet de rangement.

Cette tâche n'est pas foncièrement ardue et pour une personne quelque peu habituée au langage Smalltalk, elle peut s'avérer aisée, le langage possédant une syntaxe relativement naturelle. Toutefois, cette solution n'est pas la meilleure si l'on veut assurer une certaine viabilité au projet; peu d'analystes sont en effet censés connaître le système Smalltalk. La solution idéale serait que, lors de l'ajoût d'un nouveau type d'objets au système, l'analyste puisse déclarer les contraintes sans avoir à ce soucier du langage Smalltalk. Il introduirait ses informations soit à l'aide d'un langage de spécification, soit d'une toute autre manière (menus, choix multiples, ...). A partir de ces informations, le système générerait les expressions Smalltalk nécessaires et les insérerait dans les méthodes appropriées. De manière plus générale, ce système devrait permettre également de modifier ou supprimer des contraintes existantes et pourrait être étendu à la gestion des préconditions associées aux opérations primitives.

La réalisation d'un tel outil est tout à fait envisageable. En effet, pour chaque type d'association, il est possible de relever les différentes formes de contraintes susceptibles d'être posées sur les objets.

A chaque forme de contraintes correspond une expression Smalltalk dont le format général est toujours le même. Il suffit alors de fournir au système le format des expressions de toutes les contraintes possibles. De cette manière, le système peut

générer les expressions nécessaires et les insérer dans les méthodes appropriées.

A titre d'exemple, prenons l'ajout d'une classe Enveloppe (8.1.2.1) sous-classe de la classe Rangement. Supposons que l'on désire spécifier une contrainte associée à ce nouveau type d'objet telle qu'une instance de la classe Enveloppe ne puisse pas être composante d'un autre objet de rangement. Le format général de l'expression Smalltalk gérant cette forme de contrainte est la suivante:

une instance de la classe X ne peut être composante d'un autre objet de rangement

```
=> (self = X)
      ifTrue:[ | self error: "un objet de type X ne peut être composant d'un autre
                  objet de rangement ]
```

Cette expression teste si le receveur du message - en l'occurrence une sous-classe de la classe Rangement - est X; si c'est le cas, la méthode contenant l'expression s'arrête après avoir affiché un message d'erreur à l'écran.

En supposant que le système possède le format général de cette expression, on peut très bien envisagé qu'à l'aide des informations adéquates, il puisse générer les messages Smalltalk suivants:

```
(self = Enveloppe)
      ifTrue:[ | self error: "un objet de type Enveloppe ne peut être composant
                  d'un autre objet de rangement ]
```

et l'insère dans la méthode décrivant le message dont le sélecteur est

precondAssRgt: numCopie: unType: et: nom: bureau:

Ce principe serait le même pour les autres types d'association.

9.3.2 L'implémentation du langage de spécification

Tout au long de ce mémoire, nous avons montré qu'un analyste désireux de définir un environnement de bureau et des tâches qui lui sont associées, doit le faire grâce à des expressions Smalltalk. Chacune d'entre elles consiste à envoyer un message à une classe d'objets.

Même si le langage Smalltalk n'est pas foncièrement difficile à utiliser, un minimum d'expérience et d'habitude est demandé actuellement à un analyste. Or il n'est pas pensable d'imposer la connaissance de ce langage à tous les utilisateurs potentiels de notre logiciel. C'est pourquoi il s'avère nécessaire de mettre au point un mécanisme permettant d'utiliser le langage de spécification défini au chapitre 2. Ce mécanisme constituerait en quelque sorte un interface entre ce langage et Smalltalk.

Cet interface jouerait le rôle d'un compilateur et agirait comme suit: lorsque l'analyste déclenche l'exécution d'un texte de création d'environnement de bureau ou de tâches, celle-ci se déroulerait en deux étapes. La première consisterait à traduire le

texte en expressions Smalltalk, la seconde à exécuter le nouveau texte ainsi créé.

Un outil de ce type n'existe évidemment pas tel quel dans l'environnement Smalltalk. Pour le développer, nous devrions suivre la méthode appliquée dans le cas de la création de tout programme orienté objet. Celle-ci consiste, dans un premier temps, à recenser les grandes classes d'objets utilisées, ensuite à déterminer leurs structures de données et les messages qui leur permettront de communiquer avec d'autres objets. En tout dernier lieu, on passe au développement des méthodes décrivant les messages faisant partie de leur interface.

Au niveau des classes, nous avons besoin essentiellement de deux classes: EnvironnementCompilateur et TâcheCompilateur. La première possède toutes les informations nécessaires pour la traduction d'un texte de définition d'environnement de bureau, la seconde possède, quant à elle, les informations propres à la traduction d'un texte de définition de tâches. Chacune de ces classes aurait comme super-classe, la classe Parser, cette dernière ayant la classe Scanner pour super-classe (figure 9.2).

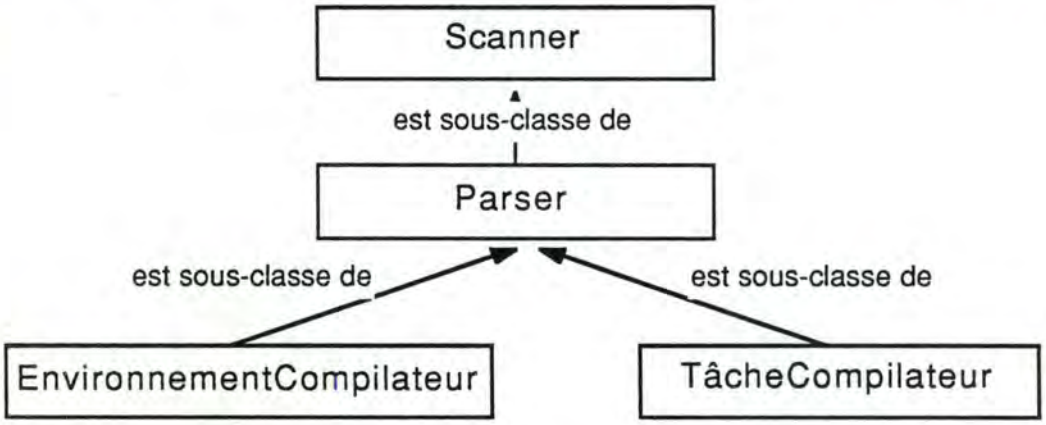


figure 9.2: Les classes EnvironnementCompilateur, TâcheCompilateur et leurs super-classes

Ces deux classes possèdent des messages et structures de données permettant l'analyse des textes et sont à la base de la conception de compilateurs et interpréteurs en Smalltalk. Les classes EnvironnementCompilateur et TâcheCompilateur héritent alors des messages qui permettent de parcourir des textes, d'analyser ces textes caractère par caractère ou suite de caractères par suite de caractères, de comparer ces suites de caractères à d'autres suites de caractères et en fonction de ces comparaisons d'effectuer telles ou telles opérations.

Le principe, appliqué par nos deux "compilateurs" afin de traduire les textes écrits dans le langage de spécification [VPMS-87] en expressions Smalltalk, serait le suivant: lorsqu'un texte leur sera soumis, ils l'analyseront séquentiellement phrase par phrase et mot par mot. A partir des informations receuillies, ils pourront générer les expressions Smalltalk adéquates.

Cependant, remarquons que, lors de la conception des deux compilateurs, il faudra veiller à ce que ceux-ci soient suffisamment souples de manière à continuer à fonctionner correctement après l'ajout par l'analyste de nouveaux types d'objets et de nouvelles opérations primitives. De manière à atteindre cet objectif, deux solutions sont

envisageables. D'une part, afin d'accepter la manipulation de nouveaux types d'objets, la syntaxe des opérations primitives et de création d'environnement décrites dans [VPMS-87] doit être revue. D'autre part, dans le but de pouvoir traduire les nouvelles opérations primitives en expressions Smalltalk, le système devrait posséder une base de données du langage contenant toutes les informations nécessaires pour générer les messages appropriés à chaque opération primitive.

9.3.3 La communication entre tâches

Dans ce travail, nous n'avons envisagé que des bureaux dans lesquels une et une seule personne travaillait et dans lesquels, à un instant donné, une et une seule tâche pouvait s'exécuter. Cependant, cela est loin de correspondre à la réalité. En effet, le travail effectué dans un bureau est souvent réalisé par différentes personnes exécutant des tâches distinctes, chacune de ces personnes contribuant à réaliser la tâche commune en exécutant sa propre tâche indépendamment des autres.

Dès lors, si nous désirons fournir un outil capable de représenter la réalité telle que nous venons de la décrire, nous devons être à même de modéliser et implémenter cette exécution de différentes tâches en parallèle.

Toutefois, deux problèmes apparaissent. Le premier a trait à l'accès partagé aux objets du bureau entre les différentes tâches. En effet, si plusieurs tâches s'exécutent parallélement dans un bureau, deux ou plusieurs de celles-ci peuvent décider au même moment d'accéder à un objet identique, d'où des possibilités de conflits à gérer. Le second problème est, quant à lui, relatif à des notions d'enchaînement et de synchronisation entre les tâches. On pourrait, par exemple, envisager qu'une tâche de bureau doive attendre l'accomplissement d'une ou de plusieurs autres tâches avant de pouvoir commencer ou poursuivre son exécution. Cette notion de synchronisation est également liée au concept d'accès partagé à un objet où une tâche doit attendre qu'une autre tâche ait libéré cet objet avant de pouvoir l'utiliser.

La réalisation de ceci en Smalltalk-80 est tout à fait possible. On y trouve, en effet, toutes les classes d'objets nécessaires. Il s'agit des classes **Process**, **ProcessorScheduler** et **Semaphore** [GOLD-83a]. Ces classes permettent d'implémenter tous les concepts de processus, sémaphore, moniteur, ... rencontrés dans le domaine de la conception de systèmes répartis et de traitement en parallèle [CORN-81], [BENA-?]. Nous allons décrire ces différentes classes et l'usage que nous en ferions afin de résoudre notre problème.

La classe **Process** fournit la description de ses instances appelées *processus*. Un processus est une séquence d'actions décrites par des expressions Smalltalk et exécutées par le système. Les actions décrites par un processus peuvent être exécutées indépendamment des actions représentées par d'autres processus. On trouve, dans le système Smalltalk, une multitude de processus. Par exemple, des processus gèrent le contrôle du clavier, de la souris, ou encore l'espace mémoire disponible.

Un processus peut prendre trois états:

- *actif:* le processus s'exécute.

- *suspendu:* le processus est interrompu et a passé la main à un autre processus.
- *terminé:* le processus a terminé son exécution.

La programmation de l'exécution des processus par le processeur de la machine Smalltalk-80 est réalisée par une instance de la classe **ProcessorScheduler**. On trouve, dans l'ensemble du système, une et une seule instance de cette classe. Elle porte le nom de *Processor* et son rôle consiste à activer et suspendre les processus en fonction des besoins et de gérer la file des processus en attente.

Généralement, la politique suivie en ce domaine est basée sur le principe du premier arrivé, premier servi (gestion de files fifo): dès que le processus actif reçoit soit le message *suspend* qui a pour objectif de le mettre dans l'état suspendu, soit le message *terminate* qui a pour objectif de le mettre dans l'état terminé, le processus qui a attendu le plus longtemps dans la file est activé. Cependant, grâce à un mécanisme de priorités associées aux différents processus, l'ordre de sélection des processus dans la file d'atttente peut être modifié.

Le parallèle entre ces processus et les tâches de bureau telles que nous les avons décrites au début de ce point 9.3.3 nous pousse à modéliser ces tâches sous forme d'une classe, soit TâcheDeBureau dont la superclasse est la classe Process (figure 9.3). Une tâche, instance de la classe TâcheDeBureau, se comporterait alors comme tout processus et pourrait s'exécuter en parallèle avec d'autres tâches.

La nouvelle classe ajouterait des méthodes et variables propres aux tâches de bureau. Parmi ces variables, on trouverait par exemple, les variables *bureau*, qui contiendrait une référence au bureau auquel une tâche est associée, et une variable *définition* contenant le texte Smalltalk la décrivant.

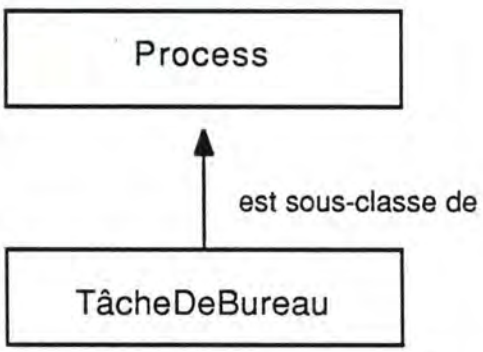


figure 9.3: La classe TâcheDeBureau et sa super-classe

Eventuellement, si cela s'avère utile, nous pourrions définir une nouvelle sous-classe de ProcessorScheduler. Cette classe fournirait la description d'un processus particulier destiné à gérer l'exécution de l'ensemble des tâches de bureau du système, instances de la classe TâcheDeBureau.

Deux problèmes restent encore à régler: l'accès partagé aux objets d'un bureau et la synchronisation entre les tâches. Il peuvent l'être grâce à la classe **Semaphore** dont les instances portent le même nom et qui sont souvent utilisées pour régler des

problèmes de mutuelles exclusions entre des processus lors de l'accès à des données partagées.

Un sémaphore peut être défini de la manière suivante: il s'agit d'un objet qui représente un "signal" ne pouvant prendre que des valeurs positives ou nulles. Généralement, les sémaphores utilisés sont des sémaphores binaires, c'est-à-dire des sémaphores ne pouvant prendre que les valeurs 0 ou 1. Un sémaphore peut recevoir les deux messages *wait* et *signal* envoyés par des processus. Ces deux messages ont les significations suivantes:

wait:	si	le sémaphore récepteur a la valeur 0
	alors	le processus émetteur est placé sur une file d'attente associé au sémaphore récepteur
	sinon	la valeur du sémaphore récepteur est décrémentée de une unité
signal:	si	la file d'attente associée au sémaphore récepteur contient des processus en attente
	alors	on en réveille un selon la politique fifo
	sinon	la valeur du sémaphore récepteur est augmentée de une unité

Dans la conception de systèmes répartis, la solution souvent choisie pour gérer l'accès à des données partagées est d'associer un sémaphore à chacune de ces données, ce sémaphore se chargeant alors de régler les conflits entre les processus demandant l'accès à la donnée.

C'est également pour cette solution que nous avons opté. A tout objet de bureau serait associé une variable supplémentaire, appelée *utilisé*, indiquant si l'objet est utilisé ou pas. Lors de la création d'un objet de bureau, une instance de la classe Semaphore y sera rangée, ce sémaphore sera un sémaphore binaire. Si la valeur de ce sémaphore est égale à 1, l'objet n'est pas utilisé par une tâche. Si, par contre, la valeur est 0, l'objet est utilisé.

Lorsqu'une tâche - un processus dans notre cas - désire utiliser un objet du bureau, elle devrait demander la permission à ce dernier à l'aide de messages appropriés. Les méthodes décrivant ces messages consiteraient notamment à envoyer notamment le message *wait* au sémaphore de l'objet demandé. Lorsque le sémaphore recevra le message *wait* envoyé par une tâche, il réagira de la manière suivante: si sa valeur est égale à 0, il place la tâche sur sa file d'attente. Si, par contre, sa valeur est égale à 1, il laisse l'accès à la tâche et met sa valeur à 0, bloquant ainsi l'accès à d'autres tâches.

Dès qu'une tâche a fini d'utiliser un objet, elle devrait prévenir ce dernier à l'aide de messages appropriés. Les méthodes déclenchées par ces messages envoient un message *signal* au sémaphore de l'objet. A ce moment, si des tâches sont en attente sur la file associée à cette dernière, le sémaphore se charge de réveiller celle qui a attendu le plus longtemps. Si, par contre, la file est vide, il remet simplement sa valeur à 1, libérant ainsi l'accès à d'autres tâches.

Ce principe des sémaphores pourrait également être appliqué pour gérer la synchronisation entre les différentes tâches par l'utilisation d'un sémaphore sur lequel

les tâches se mettraient en attente.

D'une manière plus générale, la modélisation des tâches, telle que nous l'envisageons, permettrait également de simuler à l'écran des tâches s'exécutant en parallèle mais dans des bureaux distincts. A partir de ce moment, la communication entre bureaux pourrait être simulée d'une manière plus proche de la réalité. Par exenple, supposons deux bureaux bureau1 et bureau2 disposant chacun d'une boîte In et d'une boîte Out. Un dossier, envoyé au bureau2 par le bureau1, serait transmis automatiquement de la boîte Out du bureau1 à la boîte In du bureau2.

9.3.4 Identification des objets

Nous avons vu au chapitre 8 (point 8.3) que, lorsque la fenêtre représentant le bureau est ouverte, un Pop-Up menu est associé à chaque objet. Pour que l'analyste puisse déterminer l'environnement du bureau, trois options sont disponibles dans le Pop-Up menu. Une des options permet la consultation du contenu de l'objet désigné, grâce à un répertoire. Pour chaque objet de rangement repris dans cette liste, seul le nom est donné. Pour chaque objet informationnel repris dans la liste, le nom et le numéro de copie sont fournis à l'utilisateur.

Une amélioration possible serait de pouvoir sélectionner un des objets du répertoire afin d'examiner ses attributs (nom, numéro de copie, type, ...). Un exemple est présenté figures 9.4 et 9.5.

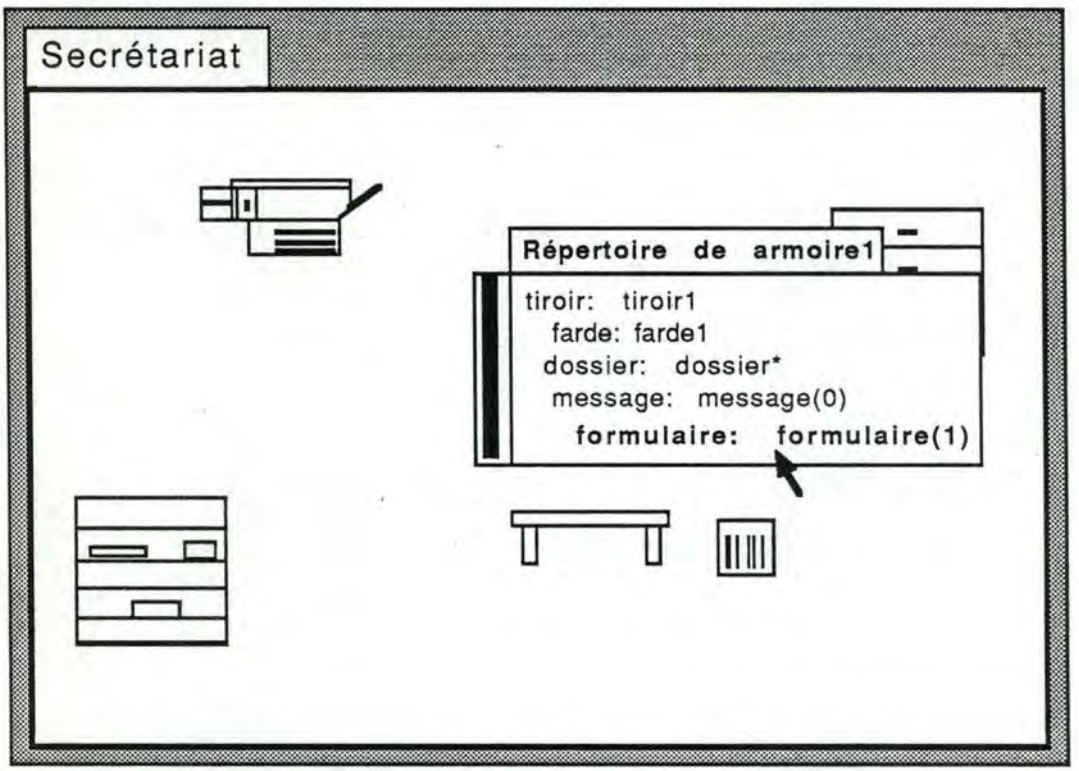


figure 9.4: Sélection d'un élément du répertoire

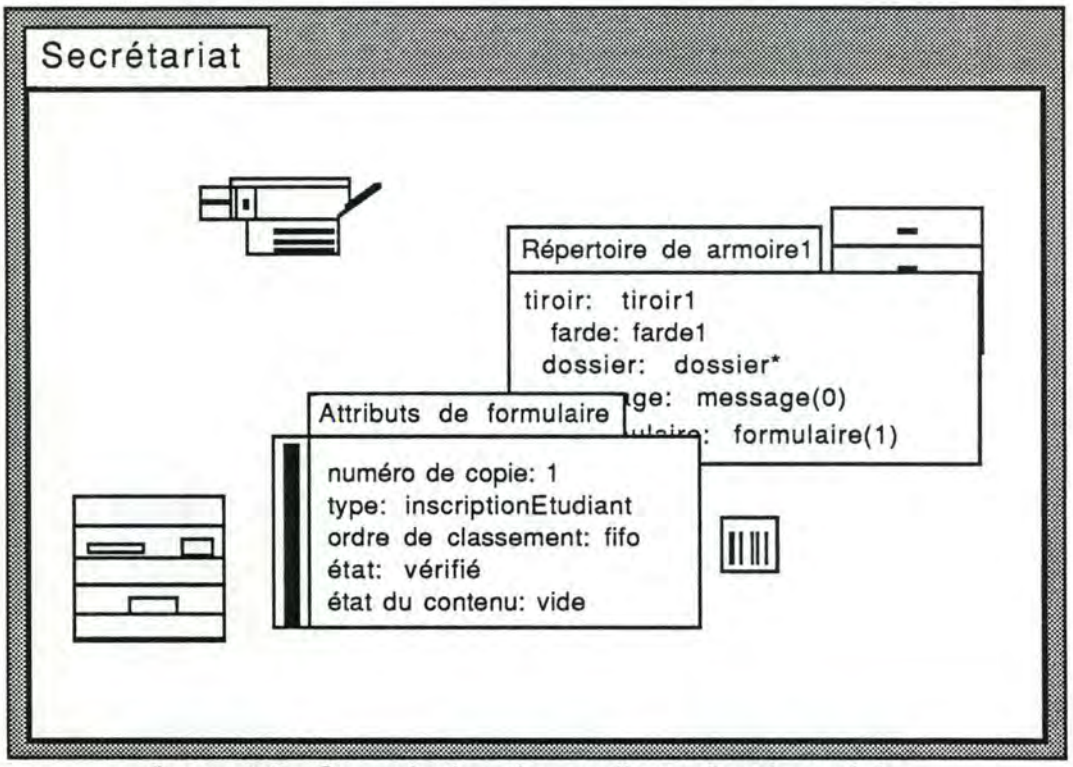


figure 9.5: Consultation des attributs d'un formulaire

9.3.5 La table de travail

La table de travail n'est considérée, dans notre logiciel, qu'en tant qu'outil de travail. Ceci ne correspond pas vraiment à la réalité car une table de travail peut servir également d'objet de rangement. En effet, une table de travail est le plus souvent conçue avec une ou plusieurs séries de tiroirs. Il serait sans doute opportun de se demander si ceux-ci ne pourraient pas appartenir à la catégorie objet de rangement. La table de travail pourrait donc cumuler les deux fonctions de traitement et de rangement de l'information.

Cette amélioration exigerait d'utiliser le principe d'héritage multiple (point 3.2.1). En effet, l'héritage multiple permet de définir une classe comme sous-classe de plusieurs super-classes. La table de travail serait alors définie comme sous-classe des classes Travail et Rangement.

Néanmoins, le mécanisme d'héritage multiple n'est pas actuellement tout à fait au point en Smalltalk-80. Il présente en effet de gros problèmes lors de la compilation. Pour cette raison, nous avons dû renoncer à lever cette contrainte.

9.3.6 Le format des objets informationnels

Lorsqu'un zoom est effectué sur un objet informationnnel lors de l'exécution d'une opération de consultation ou de vérification, un format prédéfini associé à cet objet apparaît dans une fenêtre. Ainsi, on aura un format prédéfini pour tous les formulaires, un format pour les messages et un format pour les documents. Ce format est fixe et inchangeable, ce qui signifie que, quel que soit le type d'objet informationnel réellement traité sur le bureau, le même format apparaîtra toujours à l'écran. Par

exemple, un zoom sur un formulaire de type inscriptionEtudiant affichera le même format qu'un zoom effectué sur un formulaire de type abonnement, bien que ces deux types de formulaires soient totalement différents.

Il est possible de réaliser un système qui permettrait de **modifier** le format de tout objet informationnel et celà, à l'aide d'un éditeur de texte, de la même façon que l'on peut modifier la forme d'un objet du bureau à l'aide d'un éditeur graphique. Le nouveau format serait valable pour toutes les occurrences de la classe.

Une seconde amélioration serait de permettre d'associer un format bien précis à une instance particulière d'une classe sans pour autant l'associer à toutes les instances de cette même classe. Dès ce moment, tout objet informationnel devrait, entre autres, posséder une variable d'instance supplémentaire destinée à mémoriser ce format. Le format du formulaire de type inscriptionEtudiant serait alors différent du format du formulaire de type abonnement.

9.3.7 La définition de l'environnement

La création d'un bureau et de son environnement à l'aide des méthodes Smalltalk, même si elle n'est pas vraiment compliquée, ne constitue pas une façon agréable de procéder pour un utilisateur non habitué à ce système.

Une solution plus originale et plus agréable serait la suivante: lors de la définition de l'environnement par l'utilisateur, une fenêtre représentant le bureau serait ouverte. Dans cette fenêtre, un exemplaire de chaque objet informationnel, de rangement, d'interface et de travail serait représenté. Cette série d'objets constituerait, en fait, le menu à partir duquel il serait possible de créer son environnement. L'utilisateur pourrait dupliquer chaque objet autant de fois qu'il le demanderait. Chaque nouvelle copie représenterait la création d'un nouvel objet associé au bureau. A chaque création d'un objet, l'utilisateur devrait introduire tous les renseignements permettant de l'identifier et de le qualifier. De même, les différentes associations entre objets seraient définies interactivement et de manière graphique, en le déplaçant, par exemple, sur l'objet avec lequel on veut l'associer. Cette manière de faire nous aurait permis de cacher à l'utilisateur toutes les méthodes de création d'objets et d'associations entre objets.

9.3.8 Le langage de description d'opération primitive

Nous avons laissé de côté l'implémentation du langage formalisé défini dans [VPMS-87] afin de concentrer nos efforts sur, entre-autres, le problème de l'ajout de nouveaux objets et de nouvelles opérations primitives. Cette absence du langage implique que toute l'utilisation du logiciel se fera grâce à Smalltalk-80, ce qui évidemment, n'est pas possible pour un utilisateur non informaticien. En effet, dans la version actuelle, l'ajout de nouvelles opérations primitives présente quelques difficultés dues au fait que le système est entièrement rédigé en Smalltalk et qu'une nouvelle opération primitive devra inévitablement être écrite dans le langage Smalltalk.

C'est pourquoi, il s'avérerait nécessaire de définir un langage de description d'opération primitive. Ce serait en fait un langage permettant de décrire une opération

primitive sans passer par les expressions Smalltalk. Et c'est à partir de ce langage que les instructions Smalltalk seraient générées. Une autre solution serait de fournir à l'analyste des primitives Smalltalk très simples pour qu'une utilisation de celles-ci soit possible par un utilisateur non habitué au système Smalltalk.

9.4 Conclusion

Tout au long de ce mémoire, nous avons développé un prototype d'un outil d'aide à l'automatisation du travail de bureau dans l'environnement de programmation Smalltalk-80. Toutefois, ce prototype peut être amélioré (point 9.3). En outre, grâce à la programmation orientée objet, ces modifications peuvent être apportées sans altérer le programme existant. Le code Smalltalk de ce dernier est disponible dans un volume annexe au secrétariat de l'Institut d'Informatique.

BIBLIOGRAPHIE

[ALAV-84] ALAVI, M. (1984), "An assessment of the prototyping approach to information systems development", Communications of the ACM, vol 27, n°6, pp 556-563.

[BCOX-86] COX, B.J., (1986),"Object-oriented programming, an evolutionnary approach", Addison-Wesley.

[BCOX-84] COX, B.J., (1984), "Message/Object, An Evolutionary Change", IEEE Software, January 1984, pp 50-61.

[BENA-??] BEN-ARI, M., (??), "Principles of concurrent programming", Prentice Hall International.

[BLAS-82] de BLASIS, J.P. (1982), "Les enjeux-clés de la bureautique", Les éditions d'organisation, Paris.

[BOPI-83] BODART, F., PIGNEUR, Y. (1983), "Conception assistée des applications informatiques: étude d'opportunité et analyse conceptuelle", Masson, Paris.

[BRIO-??]

[CANN-81] CANNING, R.G. (1981), "Developping systems by prototyping", EDP Analyser , vol 19, n°9, pp 1-12.

[CHEN-76] CHEN, P.P.S. (1976), "The entity-relatioship model, toward a unified view of data", ACM Transactions on Database Systems, vol 1.1, n°1, pp 9-36.

[CLAN-83] CLANTON, C. (1983), "The Future of Metaphor in Man-Computer Systems", Byte, Décembre 1983, pp 263-278.

[COLN-86] Colnet, D., Masini, G., Napoli, A., Noiret, Y., Tombre, K., (1986), "Les langages orientés objets", Centre de Recherche en Informatique de Nancy, 1986.

[CORN-81] CORNAFION, (1981), "Systèmes informatiques répartis: concepts et techniques", DUNOD informatique, BORDAS, Paris 1981.

[CUNN-85] CUNNINGHAM, W., (1985), "The Construction of Smalltalk-80 Applications", DRAFT, Computer Research Laboratory Tektronix, november 1985.

[DACH-86] DACHOUFFE, N., (1986), "Spécification et implémentation d'un modèle d'information de bureau", Mémoire inédit, Institut d'informatique, F.N.D.P., Namur.

[DIED-87] Diederich, J., Milton, J., (1987), "Experimental Prototyping in Smalltalk", IEEE Software, May 1987, pp 50-64.

[GOLD-83a] GOLDBERG, A., ROBSON, O., (1983), "Smalltalk-80, the Language and its Implementation", Addison-Wesley, 1983.

[GOLD-83b] GOLDBERG, A., (1983), "Smalltalk-80, the interactve Programming Environment", Addison-Wesley, 1983.

[GOLD-?] GOLDBERG, A., "The Influence of an Object-Oriented Language on the Programming Environment"

[HAMM-80] HAMMER, M. (1980), "What is office automation ?", Office Automation Conference, March 3-5, 1980.

[HINE-85] HINES, V. D., (1985), "Office Automation Tools and Methods for System Building", John Wiley & Sons, 1985.

[IVES-80] IVES, B. , HAMILTON, S., DAVIS, G.B. (1980), "A framework for research in computer-based management information systems", Management Science, 26, 9.

[JAMS-84] JAMSA, K.A., (1984), "Object Oriented Design vs Structure Design - Student's Perspectve", ACM SIGSOFT SOFTWARE Enginnering Notes, vol 9, n°1, January 1984, pp 43-49.

[KRAK-85] KRAKOWIAK, S. (1985), "Introduction à la Programmation par objets", méthodes et outils du génie logiciel, notes provisoires du cours de Génie Logiciel enseigné au DEA d'informatique, au DESS de Génie Informatique et à l'Ensimag.

[LUCA-78] LUCAS, H.C. (1978), "The evolution of an information system: From key-man to every-person", Sloan Management Review 19, 2, pp 39-52.

[LYYT-87] LYYTINEN, K. (1987), "Different perspectives on information systems: problems and solutions", ACM Computing Surveys, vol 19, n°1.

[MAPE-87] MAIOCCHI, R., PERNICI B. (1987), "Verification and refinement of office procedures", IEEE Computer Society Office Automation Symposium, April 1987.

[NEWM-80] NEWMAN, W. (1980), "Office Models and Office Systems Design", in "North Holland: Integrated office System-Burotics", North-Holland publishing, Company IFIP 1980.

[MEYE-87a] MEYER, B., (1987), "Object-Oriented Design and Programming", First European Software Engineering Conference, Strasbourg, 8-11, September 1987.

[MEYE-87b] MEYER, B., (1987), "Reusability: The case for Object-Oriented Design", First European Software Engineering Conference, Strasbourg, 8-11,

September 1987.

[RENT-82] RENTSCH, T., (1982), "Object Oriented Programming", ACM SIGPLAN notices, 17, 9, pp 51-57.

[ROBE-86] Robert, S., (1986), "Analyse et implémentation d'un environnement de rangement d'informations", Mémoire inédit, Institut d'informatique, F.N.D.P., Namur.

[ROBS-81] ROBSON, D., (1981), "Object-Oriented Software Systems", Byte 1981, pp 74-86.

[SMIT-77] SMITH, J.M., SMITH, D.C.P. (1977), "Database abstractions: aggregation and generalization", ACM Transactions On Database Systems, vol.2, n°2, pp 105-133.

[TESL-81] TESLER, L., (1981), "The Smalltalk Environment", Byte, August 1981, pp 90-147.

[VPMS-87] VAN PEVENAEYGE, O., SIMOENS, M. (1987), "Animation graphique de tâches de bureau", F.N.D.P., Namur.

[XRLG-81] The Xerox Learning Research Group (1981), "The Smalltalk-80 System", Byte, august 1981, pp 36-48.

ANNEXE 1

DESCRIPTION DE TACHES BUREAU A L'AIDE DU LANGAGE FORMALISE

ANNEXE 1: DESCRIPTION DE TACHES DE BUREAU A L'AIDE DU LANGAGE FORMALISE

Dans cette première annexe, nous comptons proposer trois exemples de description de tâches de bureau à l'aide du langage formalisé développé dans [VPMS-87].

Les tâches que nous allons décrire ont pour cadre général le traitement des demandes d'inscriptions de la part des étudiants auprès du secrétariat administratif de l'institut d'informatique.

1.1 Déclaration de l'environnement de bureau

Ce point reprend la définition de l'environnement du bureau dans lequel va commencer la première simulation d'une des tâches exposées dans ce qui va suivre.

```
CREER OBJET INFORMATIONNEL FORMULAIRE DE TYPE inscription IDENTIFIE PAR étudiant.
CREER OBJET INFORMATIONNEL FICHIER contacts AVEC ORDRE DE CLASSEMENT = AUCUN.
CREER OBJET INFORMATIONNEL FICHIER inscriptions AVEC ORDRE DE CLASSEMENT = AUCUN.
CREER OBJET INFORMATIONNEL FICHIER étudiants AVEC ORDRE DE CLASSEMENT = AUCUN.
CREER OBJET INFORMATIONNEL PILE pile_in AVEC ORDRE DE CLASSEMENT = LIFO.
CREER OBJET INFORMATIONNEL PILE pile_out AVEC ORDRE DE CLASSEMENT = LIFO.

CREER OUTIL DE TRAVAIL TABLE DE TRAVAIL.
CREER OUTIL DE TRAVAIL PHOTOCOPIEUSE.

CREER OBJET INTERFACE TELEPHONE.
CREER OBJET INTERFACE BOITE IN.
CREER OBJET INTERFACE BOITE OUT.
CREER OBJET INTERFACE POUBELLE.

CREER OBJET RANGEMENT ARMOIRE armoire1.
CREER OBJET RANGEMENT ETAGERE étagère1.
CREER OBJET RANGEMENT FARDE farde1.
CREER OBJET RANGEMENT TIROIR tiroir1.
CREER OBJET RANGEMENT TIROIR tiroir2.
CREER OBJET RANGEMENT TIROIR tiroir3.
CREER OBJET RANGEMENT BOITE boite1.

CREER ASSOCIATION RANGEMENT ENTRE FORMULAIRE DE TYPE inscription IDENTIFIE PAR
      étudiant ET FARDE farde1.
CREER ASSOCIATION RANGEMENT ENTRE FICHIER contacts ET TIROIR tiroir1.
CREER ASSOCIATION RANGEMENT ENTRE FICHIER inscriptions ET TIROIR tiroir2.
CREER ASSOCIATION RANGEMENT ENTRE FICHIER étudiants ET TIROIR tiroir3.

CREER ASSOCIATION INTERFACE ENTRE PILE pile_in ET BOITE IN.
CREER ASSOCIATION INTERFACE ENTRE PILE pile_out ET BOITE OUT.

CREER ASSOCIATION COMPOSITION ENTRE TIROIR tiroir1 ET ARMOIRE armoire1.
CREER ASSOCIATION COMPOSITION ENTRE TIROIR tiroir2 ET ARMOIRE armoire1.
CREER ASSOCIATION COMPOSITION ENTRE TIROIR tiroir3 ET ARMOIRE armoire1.
CREER ASSOCIATION COMPOSITION ENTRE FARDE farde1 ET BOITE boite1.
CREER ASSOCIATION COMPOSITION ENTRE BOITE boite1 ET ETAGERE étagère1.
```

1.2 Description des tâches

1.2.1 Tâche 1: Prise de contact de la part de l'étudiant

La secrétaire reçoit un contact téléphonique en provenance de l'étudiant. Elle prépare une réponse à envoyer, consistant en une lettre d'acceptation et une demande d'inscription en double exemplaires. Elle envoit la réponse à l'étudiant.

TACHE prise_contact
DESCRIPTION: traitement d'une prise de contact par la secrétaire
DECLENCHE SOUS-SCHEMA sous_schéma 1

SOUS-SCHEMA sous_schéma1
DESCRIPTION: traitement d'une prise de contact par la secrétaire
DECLENCHE PAR TACHE prise_contact
DECLENCHE OPERATION PRIMITIVE op_prim 1

OPERATION PRIMITIVE op_prim 1
DESCRIPTION: la secrétaire reçoit oralement un contact dont elle garde une trace écrite
ENONCE: RECEVOIR ORALEMENT MESSAGE AVEC TRACE ECRITE IDENTIFIEE PAR
MESSAGE contact AVEC ETAT = reçu AVEC ECHEANCE = 1987/09/01
DECLENCHEE PAR SOUS-SCHEMA sous_schéma 1
DECLENCHE OPERATION PRIMITIVE op_prim2

OPERATION PRIMITIVE op_prim 2
DESCRIPTION: la secrétaire crée un dossier vide pour la réponse
ENONCE: CREER DOSSIER DE TYPE ty_réponse IDENTIFIE PAR réponse
AVEC ETAT = créé AVEC ORDRE DE CLASSEMENT = FIFO
DECLENCHEE PAR OPERATION PRIMITIVE op_prim1
DECLENCHE OPERATION PRIMITIVE op_prim3

OPERATION PRIMITIVE op_prim 3
DESCRIPTION: la secrétaire rédige la lettre d'acceptation
ENONCE: CREER DOCUMENT DE TYPE ty_lettre IDENTIFIE PAR acceptation AVEC ETAT = créé
DECLENCHEE PAR OPERATION PRIMITIVE op_prim2
DECLENCHE OPERATION PRIMITIVE op_prim4

OPERATION PRIMITIVE op_prim 4
DESCRIPTION: la secrétaire reproduit le formulaire de demande d'inscription
vierge een deux exemplaires
ENONCE: REPRODUIRE 2 FOIS A PARTIR DU NUMERO 1
FORMULAIRE DE TYPE inscription IDENTIFIE PAR étudiant
DECLENCHEE PAR OPERATION PRIMITIVE op_prim3
DECLENCHE OPERATION PRIMITIVE op_prim5

OPERATION PRIMITIVE op_prim 5
DESCRIPTION: la secrétaire classe la lettre d'acceptaion dans le dossier réponse
ENONCE: CLASSER DOCUMENT DE TYPE ty_lettre IDENTIFIE PAR acceptation
DANS DOSSIER DE TYPE ty_réponse IDENTIFIE PAR réponse
DECLENCHEE PAR OPERATION PRIMITIVE op_prim4
DECLENCHE OPERATION PRIMITIVE op_prim6

OPERATION PRIMITIVE op_prim 6
DESCRIPTION: la secrétaire classe le premier formulaire dans le dossier réponse
ENONCE: CLASSER COPIE NUMERO 1 DE
FORMULAIRE DE TYPE inscription IDENTIFIE PAR étudiant

DANS DOSSIER DE TYPE ty_réponse IDENTIFIE PAR réponse
DECLENCHEE PAR OPERATION PRIMITIVE op_prim5
DECLENCHE OPERATION PRIMITIVE op_prim7

OPERATION PRIMITIVE op_prim 7
DESCRIPTION: la secrétaire classe le deuxième formulaire dans le dossier réponse
ENONCE: CLASSER COPIE NUMERO 2 DE
FORMULAIRE DE TYPE inscription IDENTIFIE PAR étudiant
DANS DOSSIER DE TYPE ty_réponse IDENTIFIE PAR réponse
DECLENCHEE PAR OPERATION PRIMITIVE op_prim6
DECLENCHE OPERATION PRIMITIVE op_prim8

OPERATION PRIMITIVE op_prim 8
DESCRIPTION: la secrétaire envoit le dossier réponse à l'étudiant
ENONCE: COMMUNIQUER DOSSIER DE TYPE ty_réponse IDENTIFIE PAR réponse
AVEC ETAT = envoyé AVEC ECHEANCE = 1987/09/05
DECLENCHEE PAR OPERATION PRIMITIVE op_prim7
DECLENCHE OPERATION PRIMITIVE op_prim9

OPERATION PRIMITIVE op_prim 9
DESCRIPTION: la secrétaire classe le message de contact dans le fichier des contacts
ENONCE: CLASSER MESSAGE contact DANS FICHIER contacts
DECLENCHEE PAR OPERATION PRIMITIVE op_prim8
DECLENCHE OPERATION PRIMITIVE op_prim10

OPERATION PRIMITIVE op_prim 10
DESCRIPTION: la secrétaire range le formulaire d'inscription vierge
ENONCE: RANGER FORMULAIRE DE TYPE inscription IDENTIFIE PAR étudiiant
DANS FARDE farde 1
DECLENCHEE PAR OPERATION PRIMITIVE op_prim9
DECLENCHE FIN-SOUS-SCHEMA sous_schéma 1

1.2.2 Tâche 2: Vérification et classement des demandes d'inscription

La secrétaire reçoit un dossier en provenance de l'étudiant comprenant une lettre de présentation et une demande d'inscription en double exemplaires. Après avoir consulté la lettre de présentation, elle vérifie si la demande d'inscription est complète ou non.

Si la demande n'est pas complète, elle renvoit le dossier à l'étudiant après y avoir joint une lettre de demande de renseignements.

Si la demande est complète, elle décide si l'étudiant doit ou non présenter un ou plusieurs examens d'entrée. En fonction de cette décision, elle communique à l'étudiant les documents adéquats.

La secrétaire termine par classer un exemplaire de la demande d'inscription dans le fichier des demandes d'inscription et le dossier restant dans le fichier des dossiers des étudiants.

TACHE vérification demande
DESCRIPTION: vérification par la secrétaire d'une demande d'inscription
DECLENCHE SOUS-SCHEMA sous_schéma 2

SOUS-SCHEMA sous_schéma 2
DESCRIPTION: vérification par la secrétaire d'une demande d'inscription
DECLENCHE PAR TACHE vérification demande
DECLENCHE OPERATION PRIMITIVE op_prim 11

OPERATION PRIMITIVE op_prim 11
DESCRIPTION: la secrétaire reçoit le dossier de l'étudiant par le courrier extérieur
ENONCE: RECEVOIR DOSSIER DE TYPE ty_étudiant IDENTIFIANT PAR étudiant
AVEC ETAT = reçu AVEC ECHEANCE = 1987/09/08
CONSTITUE
DE DOCUMENT DE TYPE ty_lettre IDENTIFIE PAR introduction
AVEC ETAT = reçu AVEC ECHEANCE = 1987/09/08,
DE COPIE 3 DE FORMULAIRE inscription IDENTIFIE PAR étudiant
AVEC ETAT = reçu AVEC ECHEANCE = 1987/09/08,
DE COPIE 4 DE FORMULAIRE inscription IDENTIFIE PAR étudiant
AVEC ETAT = reçu AVEC ECHEANCE = 1987/09/08
DECLENCHEE PAR SOUS-SCHEMA sous_schéma 2
DECLENCHE OPERATION PRIMITIVE op_prim 12

OPERATION PRIMITIVE op_prim12
DESCRIPTION: la secrétaire feuillète le dossier
ENONCE: FEUILLETER DOSSIER ty_étudiant IDENTIFIE PAR étudiant
AVEC ETAT= feuilleté
DECLENCHEE PAR OPERATION PRIMITIVE op_prim 11
DECLENCHE OPERATION PRIMITIVE op_prim 13

OPERATION PRIMITIVE op_prim 13
DESCRIPTION: la secrétaire vérifie la complétude du premier formulaire
ENONCE: VERIFIER COPIE NUMERO 3 DE FORMULAIRE DE TYPE inscription
IDENTIFIE PAR étudiant AVEC ETAT = vérifié
AVEC RESULTAT VERIFICATION paramètre 1
DECLENCHEE PAR OPERATION PRIMITIVE op_prim 12
DECLENCHE OPERATION PRIMITIVE op_prim 14

OPERATION PRIMITIVE op_prim14
DESCRIPTION: la secrétaire vérifie la complétude du deuxième formulaire
ENONCE: VERIFIER COPIE NUMERO 3 DE FORMULAIRE DE TYPE inscription
IDENTIFIE PAR etudiant AVEC ETAT= vérifié
AVEC RESULTAT VERIFICATION paramètre 2
DECLENCHEE PAR OPERATION PRIMITIVE op_prim 13
DECLENCHE OPERATION PRIMITIVE op_prim 15

OPERATION PRIMITIVE op_prim 15
DESCRIPTION: la secrétaire replace le premier formulaire dans le dossier
ENONCE: REPLACER COPIE NUMERO 3 DE FORMULAIRE DE TYPE inscription
IDENTIFIE PAR étudiant
DECLENCHEE PAR OPERATION PRIMITIVE op_prim 14
DECLENCHE OPERATION PRIMITIVE op_prim 16

OPERATION PRIMITIVE op_prim 16
DESCRIPTION: la secrétaire replace le deuxième formulaire dans le dossier
ENONCE: REPLACER COPIE NUMERO 4 DE FORMULAIRE DE TYPE inscription
IDENTIFIE PAR étudiant
DECLENCHEE PAR OPERATION PRIMITIVE op_prim 15
DECLENCHE CONDITION condition 1

CONDITION condition 1
DESCRIPTION: on test le résultat de la vérification du premier formulaire

ENONCE: SI PARAMETRE paramètre 1 = "INCOMPLET"
DECLENCHEE PAR OPERATION PRIMITIVE op_prim 16
SI VRAI DECLENCHE OPERATION PRIMITIVE op_prim 17
SI FAUX DECLENCHE CONDITION condition 2

CONDITION condition 2
DESCRIPTION: on test le résultat de la vérification du deuxième formulaire
ENONCE: SI PARAMETRE paramètre 2 = "INCOMPLET"
DECLENCHEE PAR CONDITION condition 1
SI VRAI DECLENCHE OPERATION PRIMITIVE op_prim 17
SI FAUX DECLENCHE OPERATION PRIMITIVE op_prim 20

OPERATION PRIMITIVE op_prim 17
DESCRIPTION: la secrétaire redige une demande de renseignements complémentaires
ENONCE: CREER DOCUMENT DE TYPE ty_lettre IDENTIFIE PAR renseignement
AVEC ETAT = créé AVEC ECHEANCE = 1987/09/12
DECLENCHEE PAR CONDITION condition 1,
PAR CONDITION condition 2
DECLENCHE OPERATION PRIMITIVE op_prim 18

OPERATION PRIMITIVE op_prim 18
DESCRIPTION: la secrétaire classe la demande de renseignements dans le dossier
ENONCE: CLASSER DOCUMENT DE TYPE ty_lettre IDENTIFIE PAR renseignement
AVEC ETAT = créé AVEC ECHEANCE = 1987/09/12
DECLENCHEE PAR OPERATION PRIMITIVE op_prim 17
DECLENCHE OPERATION PRIMITIVE op_prim 19

OPERATION PRIMITIVE op_prim 19
DESCRIPTION: la secrétaire renvoit le dossier à l'étudiant
ENONCE: COMMUNIQUER DOSSIER DE TYPE ty_étudiant IDENTIFIE PAR étudiant
AVEC ETAT = envoyé AVEC ECHEANCE = 1987/09/12
DECLENCHEE PAR OPERATION PRIMITIVE op_prim 18
DECLENCHE FIN-SOUS-SCHEMA sous_schéma 2

OPERATION PRIMITIVE op_prim 20
DESCRIPTION: la secrétaire décide si l'étudiant doit ou non passer un examen d'entrée
ENONCE: DECIDER A PARTIR DE DOSSIER ty_étudiant IDENTIFIE PAR étudiant AVEC RESULTAT
DECISION paramètre 3
DECLENCHEE PAR CONDITION condition 2
DECLENCHE CONDITION condition 3

CONDITION condition3
DESCRIPTION: on teste le résultat de la décision
ENONCE: SI PARAMETRE paramètre 3 = "EXAMEN ENTREE"
DECLENCHEE PAR OPERATION PRIMITIVE op_prim 20
SI VRAI DECLENCHE OPERATION PRIMITIVE op_prim 21
SI FAUX DECLENCHE OPERATION PRIMITIVE op_prim 23

OPERATION PRIMITIVE op_prim 21
DESCRIPTION: la secrétaire rédige le document reprenant les examens d'entrée à présenter
ENONCE: CREER DOCUMENT DE TYPE ty_lettre IDENTIFIE PAR examens
AVEC ETAT = créé AVEC ECHEANCE = 1987/09/20
DECLENCHEE PAR CONDITION condition 3
DECLENCHE OPERATION PRIMITIVE op_prim 22

OPERATION PRIMITIVE op_prim 22
DESCRIPTION: la secrétaire envoit le document à l'étudiant
ENONCE: COMMUNIQUER DOCUMENT DE TYPE ty_lettre IDENTIFIE PAR examens

AVEC ETAT = envoyé AVEC ECHEANCE = 1987/09/20
DECLENCHEE PAR OPERATION PRIMITIVE op_prim 21
DECLENCHE OPERATION PRIMITIVE op_prim 23

OPERATION PRIMITIVE op_prim 23
DESCRIPTION: la secrétaire classe le premier formulaire dans le fichier d'inscription
ENONCE: CLASSER COPIE NUMERO 3 DE FORMULAIRE DE TYPE inscription
IDENTIFIE PAR étudiant DANS FICHIER inscriptions
DECLENCHEE PAR CONDITION condition 3,
PAR OPERATION PRIMITIVE op_prim 22
DECLENCHE OPERATION PRIMITIVE op_prim 24

OPERATION PRIMITIVE op_prim 24
DESCRIPTION: la secrétaire classe le dossier dans le fichier étudiants
ENONCE: CLASSER DOSSIER DE TYPE ty_étudiant IDENTIFIE PAR étudiant
DANS FICHIER étudiants
DECLENCHEE PAR OPERATION PRIMITIVE op_prim 23
DECLENCHE FIN-SOUS-SCHEMA sous_schéma 2

1.2.3 Tâche 3: Rentrée académique

A la rentrée académique, la secrétaire reprend le fichier des demandes d'inscription, en fait une copie et transmet cette copie au secrétariat central.

TACHE rentrée académique
DESCRIPTION: la secrétaire envoit une copie du fichier d'inscriptions au secrétariat central
DECLENCHE SOUS-SCHEMA sous_schéma 3

SOUS-SCHEMA sous_schéma 3
DESCRIPTION: la secrétaire envoit une copie du fichier d'inscriptions au secrétariat central
DECLENCHE PAR TACHE rentrée académique
DECLENCHE OPERATION PRIMITIVE op_prim 25

OPERATION PRIMITIVE op_prim 25
DESCRIPTION: la secrétaire prépare un nouveau fichier d'inscriptions vide
ENONCE: CREER FICHIER duplicata AVEC ETAT = à envoyer AVEC ECHEANCE = 1987/09/26
AVEC ORDRE DE CLASSEMENT = ALPHABETIQUE
DECLENCHEE PAR SOUS-SCHEMA sous_schéma 3
DECLENCHE BOUCLE boucle1

BOUCLE boucle 1
DESCRIPTION: pour chaque formulaire du fichier d'inscription, le dupliquer en un exemplaire
ENONCE: POUR CHAQUE CONSTITUANT OIE DE FICHIER inscriptions
DECLENCHEE PAR OPERATION PRIMITIVE op_prim 25
POUR CHAQUE OCCURENCE DE variable 1 DECLENCHE SOUS-SCHEMA sous_schéma 4
ENSUITE DECLENCHE OPERATION PRIMITIVE op_prim 27

SOUS-SCHEMA sous_schéma 4
DESCRIPTION: dupliquer un formulaire
DECLENCHE PAR BOUCLE boucle 1
DECLENCHE OPERATION PRIMITIVE op_prim 26

OPERATION PRIMITIVE op_prim 26
DESCRIPTION: dupliquer un formulaire
ENONCE: REPRODUIRE 1 FOIS A PARTIR DU NUMERO 5
VARIABLE variable 1
DECLENCHEE PAR SOUS-SCHEMA sous_schéma 4

DECLENCHE FIN-SOUS-SCHEMA sous_schema 4
OPERATION PRIMITIVE op_prim27
DESCRIPTION: la secrétaire replace tous les formulaires dans le fichier d'inscription
ENONCE: REPLACER TOUS MES FORMULAIRES DE TYPE inscription
DECLRNCHEE PAR BOUCLE boucle 1
DECLENCHE BOUCLE boucle 2

BOUCLE boucle 2
DESCRIPTION: pour chaque copie effectuée, la classer dans le nouveau fichier
ENONCE: POUR CHAQUE FORMULAIRE
DECLENCHEE PAR OPERATION PRIMITIVE op_prim 27
POUR CHAQUE OCCURENCE DE variable 2 DECLENCHE SOUS-SCHEMA sous_schéma 5
ENSUITE DECLENCHE OPERATION PRIMITIVE op_prim 29

SOUS-SCHEMA sous_schema 5
DESCRIPTION: vérifier que c'est un formulaire que l'on vient de dupliquer et le classer
DECLENCHE PAR BOUCLE boucle 2
DECLENCHE CONDITION condition 4

CONDITION condition 4
DESCRIPTION: vérifier que c'est un formulaire du bon type
ENONCE: SI TYPE DE VARIABLE variable 2 EST ="inscription"
DECLENCHEE PAR SOUS-SCHEMA sous_schéma 5
SI VRAI DECLENCHE CONDITION condition 5
SI FAUX DECLENCHE FIN-SOUS-SCHEMA sous_schéma 5

CONDITION condition 5
DESCRIPTION: vérifier que c'est un formulaire avec le bon numéro de copie
ENONCE: SI NUM_COPIE DE VARIABLE variable 2 EST = "5"
DECLENCHEE PAR CONDITION condition 4
SI VRAI DECLENCHE OPERATION PRIMITIVE op_prim 28
SI FAUX DECLENCHE FIN-SOUS-SCHEMA sous_schéma 5

OPERATION PRIMITIVE OP_prim 28
DESCRIPTION: classer ce formulaire dans ce fichier
ENONCE: CLASSER VARIABLE variable 2 DANS FICHIER duplicata
DECLENCHEE PAR CONDITION condition 5
DECLLENCHE FIN-SOUS-SCHEMA sous_schéma 5

OPERATION PRIMITIVE op_prim 29
DESCRIPTION: la secrétaire envoit le fichier dupliqué au secrétariat central
ENONCE: COMMUNIQUER FICHIER duplicata AVEC ETAT = envoyé AVEC ECHEANCE = 1987/09/20
DECLENCHEE PAR BOUCLE boucle 2
DECLENCHE FIN-SOUS-SCHEMA sous_schéma 3

ANNEXE 2

DESCRIPTION D'UN ENVIRONNEMENT DU BUREAU ET DES TACHES L'AIDE DES EXPRESSIONS SMALLTALK

Annexe 2: Description d'un environnement de bureau et des tâches associées à l'aide des expressions Smalltalk

Nous allons, dans cette annexe, décrire un exemple de définition d'un environnement de bureau et des tâches associées à l'aide des méthodes Smalltalk que nous avons mises au point. Cet environnement et ces tâches ont déjà été décrits dans le langage de spécification à l'annexe 1.

Nous avons jugé bon d'insérer des commentaires afin d'expliquer ce que fait chacune des expressions. Ces commentaires apparaissent en italique entre guillemets et ne sont pas pris en compte lors de l'exécution.

2.1 La définition de l'environnement

b <- Bureau **créer:** #secrétariat.

f <- Formulaire **création:** #étudiant **numCopie:** 0 **type:** #inscription **état:** nil **échéance:** nil
ordreClassement: #aucun **bureau:** b.
fic1 <- Fichier **création:** #contacts **numCopie:** 0 **type:** nil **état:** nil **échéance:** nil
ordreClassement: #aucun **bureau:** b.
fic2 <- Fichier **création:** #inscriptions **numCopie:** 0 **type:** nil **état**: nil **échéance:** nil
ordreClassement: #aucun **bureau:** b.
fic3 <- Fichier **création:** #étudiants **numCopie:** 0 **type:** nil **état:** nil **échéance:** nil
ordreClassement: #aucun **bureau**: b.

tt <- TableTravail **créerDans:** b.
ph <- Photocopieuse **créerDans:** b.

tel <- Télephone **créerDans:** b.
bi <- BoiteIn **créerDans:** b.
bo <- BoiteOut **créerDans:** b.
po <- Poubelle **créerDans:** b.

a <- Armoire **créer:** #armoire1 **bureau:** b.
e <- Etagère **créer:** #étagère1 **bureau:** b.
f <- Farde **créer:** #farde1 **bureau:** b.
t1 <- Tiroir **créer:** #tiroir1 **bureau:** b.
t2 <- Tiroir **créer:** #tiroir1 **bureau:** b.
t3 <- Tiroir **créer:** #tiroir1 **bureau:** b.
boi <- Boîte **créer:** #boîte1 **bureau:** b.

Formulaire **associationRgtEntre:** #étudiant **numCopie:** 0 **type:** #inscription **et:** #Farde
nom: #farde1 **bureau:** b.
Fichier **associationRgtEntre:** #contacts **numCopie:** 0 **type:** nil **et:** #Tiroir **nom:** #tiroir1
bureau: b.
Fichier **associationRgtEntre:** #inscriptions **numCopie:** 0 **type:** nil **et:** #Tiroir **nom**: #tiroir2
bureau: b.
Fichier **associationRgtEntre:** #étudiant **numCopie:** 0 **type:** #inscription **et:** #Tiroir
nom: #tiroir3 **bureau:** b.

Tiroir **associationCompEntre:** #tiroir1 **et:** #Armoire **nomComposé:** #armoire1 **bureau:** b.
Tiroir **associationCompEntre:** #tiroir2 **et:** #Armoire **nomComposé:** #armoire1 **bureau:** b.
Tiroir **associationCompEntre:** #tiroir3 **et:** #Armoire **nomComposé:** #armoire1 **bureau:** b.
Farde **associationCompEntre:** #farde1 **et:** #Boîte **nomComposé:** #boîte1 **bureau:** b.
Boîte **associationCompEntre:** #boîte1 **et:** #Etagère **nomComposé:** #étagère1 **bureau:**b.

2.2 La définition des tâches

2.2.1 Tâche 1 : Prise de contact de la part de l'étudiant

Msge **recevoirOralement:** #contact **type:** nil **état:** #reçu **échéance:** '09/01/87' **bureau:** b.

"*La secrétaire reçoit oralement un contact dont elle garde la trace écrite* "

Dossier **créer:** #réponse **numCopie:** 0 **type:** #typeRéponse **état:** #créé **échéance:** nil **ordreClassement:** #fifo **bureau:** b.

"*La secrétaire crée un dossier vide pour répondre à l'étudiant* "

Document **créer:** #acceptation **numCopie:** 0 **type:** #inscription **quantité:** 2 **àPartirDe:** 1 **bureau:** b.

"La secrétaire reproduit en deux exemplaires un formulaire de demande d'inscription vierge"

Document **classer:** #acceptation **numCopie:** 0 **type:** #typeLettre **dans:** #Dossier **oisNom:** #réponse **oisType:** #typeRéponse **bureau:** b.

"La secrétaire classe la lettre d'acceptation dans le dossier de réponse"

Formulaire **classer:** #étudiant **numCopie:** 1 **type:** #inscription **dans:** #Dossier **oisNom:** #réponse **oisType:** #typeRéponse **bureau:** b.

" *La secrétaire classe le premier formulaire dans le dossier de réponse"*

Formulaire **classer:** #étudiant **numCopie:** 2 **type:** #inscription **dans:** #Dossier **oisNom:** #réponse **oisType:** #typeRéponse **bureau:** b.

" *La secrétaire classe le deuxième formulaire dans le dossier de réponse"*

Dossier **communiquer:** #réponse **numCopie:** 0 **type:** #typeRéponse **état:** #envoyé **échéance:** '09/05/1987' **bureau:** b.

"La secrétaire envoit le dossier de réponse à l'étudiant"

Msge **classer:** #contact **numCopie:** 0 **type:** nil **dans:** #Fichier **oisNom:** #contacts **oisType:** nil **bureau:** b.

"La secrétaire classe la trace écrite du contact dans le fichier des contacts"

Formulaire **ranger:** #étudiant **numCopie:** 0 **type:** #inscription **dans:** #Farde
nom: #farde1 **bureau:** b.

"La secrétaire range le formulaire de demande d'inscription vierge"

2.2.2 Tâche 2 : Vérification et classement des demandes d'inscription

Dossier **recevoirOis**: #étudiant **numCopie:** 0 **type:** #typeEtudiant **état:** #reçu
échéance: '9/8/87' **ordreClassement:** #aucun
constituéDe: #(Document #introduction 0 #typeLettre #reçu '09/08/87'
Formulaire #étudiant 3 #inscription #reçu '09/08/87'
Formulaire #étudiant 4 #inscription #reçu '09/08/87')

"*La secrétaire reçoit par le courrier extérieur le dossier de l'étudiant*"

Dossier **feuilleter:** #étudiant **type:** #typeEtudiant **ordreFeuilletage:** #aucun **bureau:** b.

"*La secrétaire feuillète le dossier*"

par1 <- Formulaire **vérifier:** #étudiant **numCopie:** 3 **type:** #inscription **bureau:** b.

" *La secrétaire vérifie la complétude du premier formulaire* "

par2 <- Formulaire **vérifier:** #étudiant **numCopie:** 4 **type:** #inscription **bureau:** b.

" *La secrétaire vérifie la complétude du deuxième formulaire* "

Formulaire **replacer:** #étudiant **numCopie:** 3 **type:** #inscription **bureau:** b.

" *La secrétaire replace le premier formulaire dans le dossier* "

Formulaire **replacer:** #étudiant **numCopie:** 4 **type:** #inscription **bureau:** b.

" *La secrétaire replace le deuxième formulaire dans le dossier* "

par1 = #INCOMPLET
ifTrue: [Document **créer:** #renseignement **numCopie:** 0 **type:** #typeLettre
état: #créé **échéance:** '9/12/87' **ordreClassement:** #aucun
bureau: b.

"*La secrétaire rédige une demande de renseignements complémentaires*"

Document **classer:** #renseignement **numCopie:** 0 **type:** #typeLettre
dans: #Dossier **oisNom:** #étudiant **oisType:** #typeEtudiant
bureau: b.

"*La secrétaire classe la demande de renseignements dans le dossier*"

Dossier **communiquer:** #étudiant **numCopie:** 0 **type:** #typeEtudiant
bureau: b

"*la secrétaire renvoit le dossier à l'étudiant*"]

ifFalse: [(par2 = #INCOMPLET)
ifTrue: [Document **créer:** #renseignement **numCopie:** 0
type: #typeLettre **état:** #créé **échéance:** '9/12/87'
ordreClassement: #aucun **bureau:** b.

"*La secrétaire rédige une demande de renseignements complémentaires*"

Document **classer:** #renseignement **numCopie:** 0
type: #typeLettre **dans:** #Dossier **oisNom:** #étudiant
oisType: #typeEtudiant **bureau:** b.

"*La secrétaire classe la demande de renseignement dans le dossier*"

Dossier **communiquer:** #étudiant **numCopie:** 0
type: #typeEtudiant **bureau:** b

"*la secrétaire renvoit le dossier à l'étudiant*"]

ifFalse: [par3 <- Dossier **déciderAPartirDe:** #étudiant **numCopie:** 0
type: #typeEtudiant **bureau:** b.

"*La secrétaire décide si l'étudiant doit ou nom passer des examens d'entrée*"

par3 = #ACCEPTABLE
ifTrue: [Formulaire **classer:** #étudiant **numCopie:** 3
type: #inscription **dans:** #Fichier
oisNom:#inscriptions **oisType:** nil
bureau: b.

"*La secrétaire classe le premier formulaire dans le fichier des demandes d'inscription*"

Dossier **classer:** #étudiant **numCopie:** 0
type: #typeEtudiant **dans:** #Fichier
oisNom: #étudiants **oisType:** nil
bureau: b

"*La secrétaire classe le dossier dans le fichier d'étudiants*"]

ifFalse: [Document **créer:** #examens **numCopie:** 0
type: #typeLettre **état:** #créé
échéance: '9/20/87'
ordreClassement: #aucun
bureau: b.

"*La secrétaire rédige le document reprenant les examens d'entrée à présenter*"

Document **communiquer:**#examens
numCopie: 0 **type:** #typeLettre **bureau:** b.

"*Le secrétaire envoit le document à l'étudiant*"

Formulaire **classer:** #étudiant **numCopie:** 3
type: #inscription **dans:** #Fichier
oisNom:#inscriptions **oisType:** nil
bureau: b.

"*La secrétaire classe le premier formulaire dans le fichier des demandes d'inscription*"

Dossier **classer:** #étudiant **numCopie:** 0
type: #typeEtudiant **dans:** #Fichier
oisNom: #étudiants **oisType:** nil
bureau: b

"*La secrétaire classe le dossier dans le fichier d'étudiants*"]]]

2.2.3 Tâche 2 : Rentrée académique

Fichier **créer:** #duplicata **numCopie:** 0 **type:** nil **état:** #àEnvoyer **échéance:** '9/26/87'
ordreClassement: #alphabétique.

"*La secrétaire prépare un nouveau fichier d'inscriptions vide*"

(Information **tous:** #oie **de:** #Fichier **nom:** #inscriptions **type:** nil **dans:** b)

"*Pour chaque objet élémentaire du fichier inscriptions*"

do: [: variable1 I Information **reproduire:** variable1 **quantité:** 1 **àPartirDe:** 5
bureau:b.]

"*Reproduire l'objet élémentaire en un exemplaire*"

Formulaire **replacerTous:** #inscription **bureau:** b.

"*La secrétaire replace tous les formulaires qui ont été dupliqués*"

(Information **tous:** #Formulaire **dans:** b)

"*Pour tous les formulaires du bureau*"

do: [: variable2 I (Information **valeurAttributDe:** variable2 **attribut:** #type **bureau:** b)
= #inscription

"*vérifie que c'est un formulaire de demande d'inscription*"

ifTrue: [(Information **valeurAttributDe:** variable2 **attribut:** #numCopie
bureau: b) = 5

"*vérifie que c'est une copie à envoyer*"

ifTrue: [Information **classer:** variable2 **dans:** #Fichier
oisNom: #duplicata **oisType:** nil **bureau:** b]].

"*classe le formulaire dans le nouveau fichier*"

Fichier **communiquer:** #duplicata **numCopie:** 0 **type:** nil **état:** #envoyé
échéance: '9/20/87' **bureau:** b.

"*La secrétaire envoit le fichier dupliqué au secrétariat central*"

ANNEXE 3

LE MODE D'EMPLOI DU LOGICIEL

ANNEXE 3 : LE MODE D'EMPLOI DU LOGICIEL

Avertissements:

1) Avant de décrire le mode d'emploi de notre logiciel, rappelons qu'il a été entièrement réalisé dans un environnement Smalltalk. La machine utilisée est une station de travail Tektronix de la série 440X et le programme ne peut être installé sur une autre machine. En outre, faute de temps, l'interface entre le langage de spécification d'environnement et de tâches, décrit au chapitre 2, et les expressions Smalltalk n'a pu être implémenté. Ceci a pour conséquence immédiate que la définition d'un bureau et des tâches associées ne peut se faire que dans la syntaxe Smalltalk-80. Actuellement, il est donc requis que l'analyste possède quelques connaissances de base en ce domaine pour faire une utilisation valable du logiciel. Toutefois, cette syntaxe étant relativement naturelle, cela ne devrait pas constituer un obstacle trop important.

2) Comme l'environnement Smalltalk, notre logiciel utilise abondamment les fenêtres et la souris. Ces éléments sont organisés selon l'architecture Modèle-Vue-Contrôleur expliquée dans le chapitre 4.

   La souris utilisée possède trois boutons qui peuvent prendre différentes significations selon la fenêtre où on se trouve. De manière générale:

   - le bouton de *gauche* permet de faire des sélections de texte à l'écran.
   - le bouton du *centre* permet d'accéder à un Pop-Up menu dont les options sont spécifiques d'un type de fenêtre à un autre et d'en choisir une des options.
   - le bouton de *droite* fait apparaître également un Pop-Up menu qui regroupe, quant à lui, des fonctions communes à toutes les fenêtres.

     Exemple: fermer, rétrécir, déplacer une fenêtre ... .

   Dans les figures qui vont être citées et qui sont rassemblées en fin de cette annexe, la souris sera représentée par un rectangle composé de trois autres rectangles correspondant chacun à un des boutons de la souris. Un de ces rectangles noirci signifie que le bouton correspondant est pressé.

Une fois rentré dans le système Smalltalk, l'analyste doit copier sur le disque les différents fichiers contenant les classes d'objets qui forment l'ensemble du programme. Ces fichiers sont **BureauClasses.st**, **BureauInterface.st**, **BureauInfo.st**, **BureauObjets.st**.

Accessoirement, on peut également y copier les fichiers secretariat, secretariatTache1, secretariatTache2 et secretariatTache3. Ces fichiers constituent un exemple de démonstration et contiennent le texte Smalltalk de la description d'un environnement de bureau et de trois tâches associées à ce bureau. Quand les fichiers ont été copiés, il reste à les charger dans le système Smalltalk. Pour toutes ces opérations, nous renvoyons le lecteur au manuel d'utilisation des stations de travail

Tektronix.

Lorsque tout ceci a été réalisé, l'analyste peut décider de faire trois choses:

- simuler une tâche dans un bureau donné.
- ajouter/supprimer une classe d'objets dans la hiérarchie existante.
- ajouter/supprimer une opération primitive.

1. La simulation d'une tâche

La simulation d'une tâche se fera en deux étapes:

- la définition de l'environnement du bureau et le placement des objets dans la fenêtre représentant ce bureau.
- la définition et la simulation d'une tâche.

1.1 La définition de l'environnement de bureau

1.1.1 La description de l'environnement

Pour décrire un environnement de bureau, l'analyste doit procéder comme suit:

- Il exécute, dans un workspace, l'expression Smalltalk **Bureau definitionEnvironnement**.

Pour cela, il sélectionne cette expression et choisit l'option **do it** dans le Pop-Up menu associé au bouton central de la souris (figure 3.1).

- L'exécution de cette expression a pour effet d'ouvrir une fenêtre dans laquelle la question "Nom du bureau?" apparait.

- L'analyste introduit le nom du bureau dont il veut définir l'environnement (figure 3.2) et ferme la fenêtre à l'aide soit de l'option **close** du Pop-Up menu associé au bouton droit de la souris, soit de la touche **return** du clavier.

- Le programme va alors rechercher sur le disque, un fichier dont le nom correspond au nom de bureau introduit et qui doit contenir une éventuelle définition de bureau. Deux situations peuvent se présenter:

1) Il existe une définition pour ce bureau et le programme a trouvé le fichier correspndant. Il ouvre alors une fenêtre contenant cette définition. (figure 3.3)

2) Le programme n'a pas trouvé le fichier, ce qui signifie que le bureau n'a pas encore été défini auparavant. Si c'est le cas, le programme demande à l'utilisateur s'il faut le créer ou pas (figure 3.4).

   Pour répondre par l'affirmative, l'analyste choisit l'option **proceed** dans le menu associé au bouton du centre de la souris. Sinon, il ferme la fenêtre à

l'aide de l'option **close** présente dans le Pop-Up menu associé au bouton de droite de la souris et le programme s'arrête. S'il a choisi de créer le fichier, le système ouvre une nouvelle fenêtre contenant la phrase " -NOUVEAU BUREAU- ".

- Quand la fenêtre destinée à contenir la description du bureau est ouverte, l'analyste peut soit modifier, soit composer son texte Smalltalk de description d'environnement. Il introduit son texte au clavier et peut utiliser les options classiques de tout éditeur de texte *again*, *undo*, *copy*, *cut* et *paste* du menu associé au bouton du centre de la souris.

Le texte de la définition doit toujours suivre le schéma suivant:

- la création d'une instance de la classe Bureau.
- la création des différents objets de ce bureau et des associations entre eux.
- l'envoi du message **ouvrirDecor** à l'instance de la classe Bureau. L'exécution de cet envoi a pour but de vérifier si toutes les contraintes d'intégrité sont vérifiées (exemple: il doit y avoir une table de travail dans le bureau) et d'ouvrir une fenêtre représentant le bureau et ses objets.

- Lorsqu'il est satisfait de son texte, l'analyste peut décider de le sauver et de le conserver sur le disque. Il lui suffit alors de sélectionner l'option **save** du Pop-Up menu associé au bouton central de la souris (figure 3.5). Il est à souligner que si une version précédente de la définition existe, elle est remplacée par la nouvelle.

1.1.2 La création du bureau et de son décor

- Dès que son texte de description d'environnement de bureau est terminé, l'analyste peut réellement créer son bureau et ouvrir une fenêtre le représentant et contenant ses objets.Il pourra ainsi composer le décor dans lequel vont se dérouler les simulations.

- Pour cela, l'analyste doit sélectionner et exécuter chacune des expressions de son texte. Il peut l'exécuter groupe d'expressions par groupe d'expressions, ou d'un bloc (figure 3.6)

- Lors de l'ouverture initiale de la fenêtre du bureau ne sont présentés que les objets qui ne sont ni rangés, ni classés, ni communiqués dans un autre objet ou qui ne composent aucun autre objet. Ils sont affichés les uns à la suite des autres de gauche à droite et de haut en bas dans la fenêtre du bureau. (figure 3.7).

- La fenêtre du bureau étant ouverte, l'analyste peut avoir accès à deux types de Pop-Up menu. Le premier est associé à chaque objet présenté et le second est associé à la fenêtre du bureau.

- A chaque objet affiché est associé un Pop-Up menu. Pour y accéder, il suffit de positionner le curseur sur l'objet désiré et de presser le bouton central de la souris (figure 3.8).

Ce Pop-Up menu offre trois options:

1) **déplacer**: cette option permet le déplacement, à l'aide de la souris, d'un objet dans les limites de la fenêtre du bureau. L'objet se déplace suivant les mouvements de la souris jusqu'à ce qu'un bouton de cette dernière soit pressé. A ce moment, l'objet se fixe à l'endroit indiqué par le curseur. Si on tente de faire sortir l'objet de la fenêtre du bureau, il reprend automatiquement sa position initiale. L'analyste peut ainsi déplacer chaque objet de manière à représenter son bureau de la manière la plus proche possible. Les positions des objets sont mémorisées en vue de la simulation des tâches.Toutefois, une contrainte subsiste encore à l'heure actuelle. En effet, lorsque l'on désire déplacer un objet, tous les objets qui le chevauchent doivent être déplacés auparavant.

2) **identifier:** cette option affiche le nom de l'objet à partir duquel on a accédé au Pop-Up menu (figure 3.9). Ce nom reste affiché jusqu'à ce qu'un bouton de la souris soit pressé. Cette option est très utile car, pour une classe d'objets, le dessin qui lui est associé est identique pour toutes ses instances. Dès lors, si un bureau comporte deux étagères, il est impossible de les distinguer à l'oeil nu. Le fait de pouvoir obtenir leur nom permet de les identifier.

3) **consulter:** cette option est très interessante également car elle permet d'obtenir la liste des objets se trouvant dans ou sur l'objet à partir duquel on a accédé au Pop-Up menu. Cette liste constitue un répertoire hiérarchique de l'objet. Afin d'illustrer cela, prenons un exemple: supposons qu'on dispose d'une armoire de nom *armoire* qui contient les tiroirs de nom *tiroir1*, *tiroir2* et *tiroir3.* Le premier tiroir contient deux fardes de nom *farde1* et *farde2.* La farde *farde1* contient le dossier de nom *dossier.* Ce dossier renferme le message de nom *message* et de numéro de copie *0*, et le formulaire de nom *formulaire* et de numéro de copie *1*. Le répertoire se présente comme ceci:

   Répertoire de armoire

   ```
   tiroir: tiroir1
     farde: farde1
       dossier: dossier *
         message: message (0)
           formulaire: formulaire (1)
       farde: farde2
   tiroir: tiroir2
   tiroir: tiroir3
   ```

   L'astérisque indique que le dossier et ses composants ne sont pas physiquement dans la farde pour le moment. A l'écran, ce répertoire sera présenté dans une fenêtre (figure 3.10).

   Etant données les contraintes posées sur les objets affichés, la farde et ses composants ne sont pas représentés à l'écran mais le répertoire permet de

s'assurer de leur présence dans le bureau.

Dès que la fenêtre du bureau est ouverte, l'analyste peut ,à tout moment, décider de définir ou de simuler une tâche dans le bureau. C'est ce que nous allons voir au point suivant.

1.2 La définition et l'exécution d'un tâche de bureau

- La définition d'une tâche, dans le bureau analysé, se fera au moyen du Pop-Up menu de la fenêtre du bureau. ce menu est accessible par pression du bouton du milieu de la souris lorsque le curseur se trouve dans le bureau et qu'il ne désigne aucun des objets. Il possède deux options: **tâche** et **tâchesListe** (figure 3.11).

- L'option **tâchesListe** ouvre une fenêtre contenant la liste des tâches associées au bureau représenté (figure 3.12).

Dans la figure 3.12, trois tâches ont été déjà été définies par l'analyste.

La fenêtre peut être fermée grâce à l'option **close** du Pop-Up menu du bouton de droite de la souris.

- Pour accéder à une tâche particulière ou pour en créer une nouvelle, l'analyste choisit l'option **tâche** dans le Pop-Up menu de la fenêtre du bureau. A ce moment, apparait une nouvelle fenêtre dans laquelle est affichée la question **Nom de la tâche?** (figure 3.13).

- L'analyste introduit le nom de la tâche et ferme la fenêtre à l'aide soit de la touche **return** du clavier, soit de l'option **close** du Pop-Up menu du bouton de droite de la souris. Cela a pour effet de rechercher sur le disque le fichier dans lequel se trouve le texte Smalltalk de la tâche spécifiée.

Deux situations peuvent alors se présenter:

1) La tâche a déjà été définie pour le bureau dont la fenêtre est ouverte et le système a trouvé le fichier. Il ouvre une nouvelle fenêtre contenant le texte de cette tâche (figure 3.14).

2) La tâche n'a pas encore été définie pour le bureau dont la fenêtre est ouverte. Le système ouvre alors une fenêtre contenant la phrase -NOUVELLE TACHE- (figure 3.15).

C'est à partir de ce moment que commence la définition proprement dite.

- L'utilisateur peut modifier ou éditer du texte dans la fenêtre. Ce texte est sensé représenter la description d'une tâche de bureau. Il dispose de quelques fonctions propres à tout éditeur de texte: *again*, *undo*, *cut*, *paste* et *copy* . Ces fonctions sont associées au bouton central de la souris.
- Lorsqu'il est satisfait du texte Smalltalk qu'il a composé, l'analyste peut le sauver et le conserver sur disque. Il lui suffit de sélectionner l'option save du Pop-Up

menu associé au bouton central de la souris (figure 3.16). La tâche désignée est alors associée au bureau sous analyse et on ne peut y accéder que par l'intermédiaire de ce dernier. Il est à souligner que si une version précédente de la tâche existait, elle est remplacée par la nouvelle.

- Pour lancer la simulation de la tâche, il lui suffit de sélectionner le texte la décrivant et de choisir l'option **do it** dans le Pop-Up menu associé au bouton central de la souris.

Toutefois, un problème subsiste actuellement. En effet, le texte de la définition de la tâche doit être exécuté dans la même fenêtre que l'a été le texte de la définition de l'environnement. En effet, les opérations primitives composant la tâche utilisent l'instance du bureau comme argument. Or cette instance, ainsi que les objets qui lui sont associés, sont liés à la fenêtre dans laquelle ils ont été créés. C'est pourquoi, avant d'exécuter la tâche, l'analyste doit recopier le texte la décrivant dans la fenêtre de création de l'environnement. Pour cela, il sélectionne le texte et choisit l'option **copy** du menu du bouton central de la souris. Ensuite, il positionne son curseur dans la fenêtre où doit être recopié le texte et choisit l'option **paste** du bouton central de la souris (figure 3.17).

Alors seulement, il peut exécuter sa tâche.

remarque:

Afin de pouvoir identifier correctement les objets lors de la simulation, leur nom est affiché à l'écran pendant quelques secondes dès qu'ils doivent être manipulés par la tâche. Ainsi, si un dossier sort d'une étagère pour se déplacer dans le bureau, le nom de l'étagère et du dossier s'imprimeront à l'écran avant que l'objet informationnel ne commence à se mouvoir.
(figure 3.18)

2 L'ajout d'un nouveau type d'objets

Comme nous l'avons expliqué au chapitre 8, ajouter un nouveau type d'objets consiste à ajouter une nouvelle classe comme sous-classe d'une autre classe.

Pratiquement, l'ajout d'un nouveau type d'objets se déroulera de la manière suivante:

- L'utilisateur exécute, dans une vue Workspace, l'expression Smalltalk **Bureau hiérarchie** (figure 3.19). L'exécution de cette expression a pour effet d'ouvrir une fenêtre composée de deux parties distinctes (figure 3.20).

La première partie contient les noms de tous les types d'objets de bureau définis. Afin de voyager dans la liste, il suffit de déplacer le rectangle noir - appelé scrollBar - qui apparait à gauche lorsque le curseur est positionné dans la fenêtre (figure 3.21).
La seconde partie est destinée à l'affichage de texte.

- Cliquer sur un des noms de la liste permet d'afficher, dans la seconde partie de la fenêtre, la hiérarchie des objets comprenant l'objet sélectionné (figure 3.22). L'affichage d'une hiérarchie pour n'importe quel type d'objet donne à l'utilisateur la possibilité de bien situer l'objet qu'il veut ajouter, par rapport aux autres types d'objets existants.

- Lorsqu'un nom est sélectionné dans la partie supérieure de la fenêtre, un Pop-Up menu est accessible grâce au bouton central de la souris (figure 3.23). Ce menu possède quatre options:

1) **add an object**: cette option permet de créer une nouvelle classe dont la super-classe est la classe sélectionnée. La nouvelle classe hérite donc de toutes les propriétés de cette dernière classe et de ses super-classes. Lorsque l'utilisateur choisit cette option, une nouvelle fenêtre s'ouvre pour demander le nom du nouveau type d'objets (figure 3.24). Ce nom doit commencer par une majuscule mais le système le corrige automatiquement si ce n'est pas le cas. Lorsqu'il est créé, le nouveau type d'objets reçoit automatiquement une forme prédéfinie pour éviter à l'utilisateur de devoir le dessiner immédiatement.

2) **change the form**: cette option permet de modifier la forme associée au type d'objets sélectionné. Le choix de cette option ouvre un éditeur graphique contenant la forme actuelle du type d'objet. L'utilisateur peut la modifier comme bon lui semble.

3) **remove**: le choix de cette option permet de supprimer la classe d'objets sélectionnée.

4) **rename**: le choix de cette option permet de renommer la classe d'objets sélectionnée.

3. L'ajout d'opérations primitives

Afin de décrire les tâches de bureau qu'il désire simuler à l'écran, l'analyste utilise des expressions Smalltalk. Ces expressions consistent à envoyer des messages à la classe Information ou à une de ses sous-classes. Elles décrivent les opérations primitives et les structures d'enchaînement qui composent la tâche. Tous ces messages ont été définis dans le protocole de la classe Information.

Cependant, lors de l'analyse d'un bureau particulier, l'analyste peut estimer avoir besoin d'une opération primitive supplémentaire. S'il désire ajouter cette opération primitive au système, il devra l'insérer dans le protocole de classe de la classe Information. Cela peut être réalisé à l'aide des outils faisant partie de l'environnement de programmation Smalltalk tels que les System Browser. Toutefois, dans le but de faciliter son travail, nous mettons à la disposition de l'analyste un outil plus simple d'emploi.

Pratiquement, l'ajout d'une nouvelle opération primitive se déroulera de la manière suivante:

- L'analyste exécute, dans une vue Workspace, l'expression Smalltalk **bureau ajoutOperation** (figure 3.24).

- L'exécution de cette expression a pour effet d'ouvrir une fenêtre dans laquelle apparait six autres fenêtres (figure 3.25):

  - La première [1] est destinée à contenir une liste de classes. Dans le cas de notre outil, seule la classe **Information** apparait.

  - La seconde [2] contient la liste des catégories de messages appartenant au protocole de la classe Information. Dans le cas de notre outil, seuls les protocoles **simulation** et **precondition** apparaissent.

  - La troisième [3] présente la liste des sélecteurs des messages appartenant à la catégorie de messages sélectionnée dans la fenêtre [2].

  - La quatrième [4] est une fenêtre permettant d'éditer du texte. C'est dans cette fenêtre que l'utilisateur introduira le texte Smalltalk de sa nouvelle opération primitive.

  - Quant aux deux cases **instance/class**, elles permettent de sélectionner soit le protocole de classe, soit le protocole d'instance de la classe affichée dans la fenêtre [1]. Dans le dessin de la figure 3.25, le protocole de classe a été sélectionné.

- Pour ajouter une opération primitive, l'analyste sélectionne la catégorie de messages simulation et ne sélectionne aucun sélecteur dans la vue [3]. A ce moment, apparait, dans la vue [4], le format général d'un message (figure 3.26).

- Dès ce moment, l'analyste peut introduire, dans la vue le texte, de la méthode décrivant sa nouvelle opération primitive [4].

- Lorsqu'il est satisfait de son texte il peut le compiler, le sauver et l'insérer dans le protocole de classe de la classe information. Il fait cela en sélectionnant l'option **accept** dans le Pop-Up menu associé à la fenêtre [4]. Si des erreurs se sont glissées dans son texte, le système Smalltalk-80 se chargera de prévenir l'analyste afin que celui-ci les corrige.

- Sélectionner la catégorie de messages précondition permet d'ajouter un message destiné à la vérification des préconditions associées à une opération primitive. La gestion des préconditions se fera de la manière présentée au chapitre 9.

- Sélectionner un sélecteur dans la vue [3] permet d'afficher, dans la fenêtre [4], le texte de la méthode décrivant le message identifié par ce sélecteur.

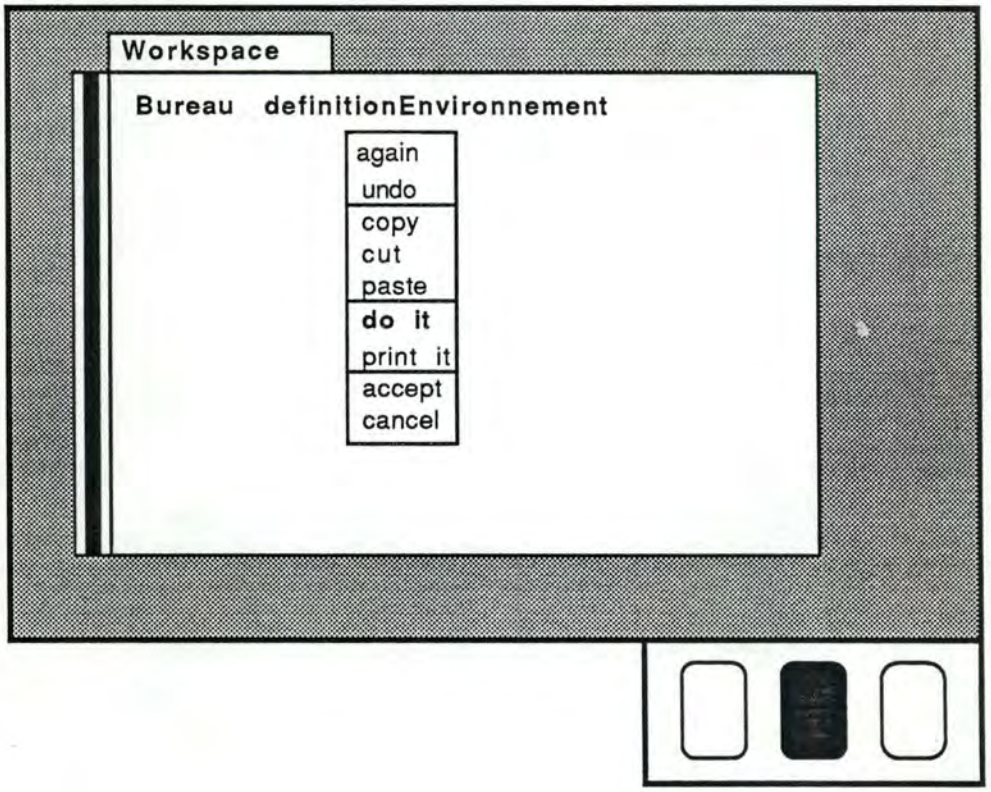


figure 3.1

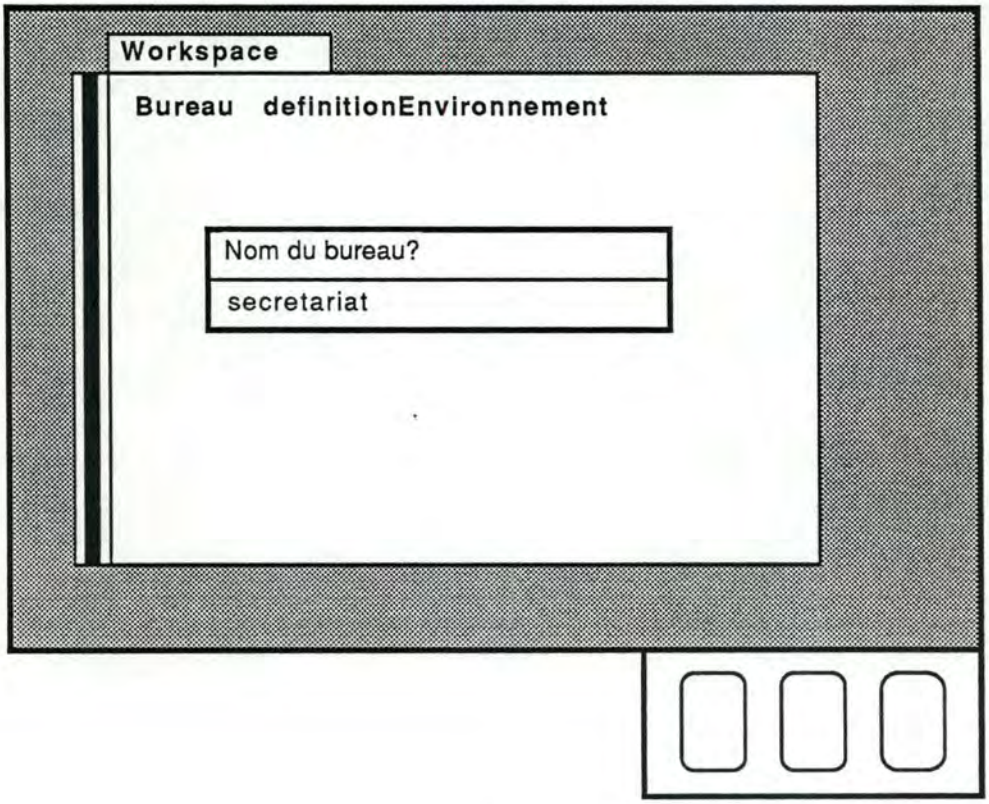


figure 3.2

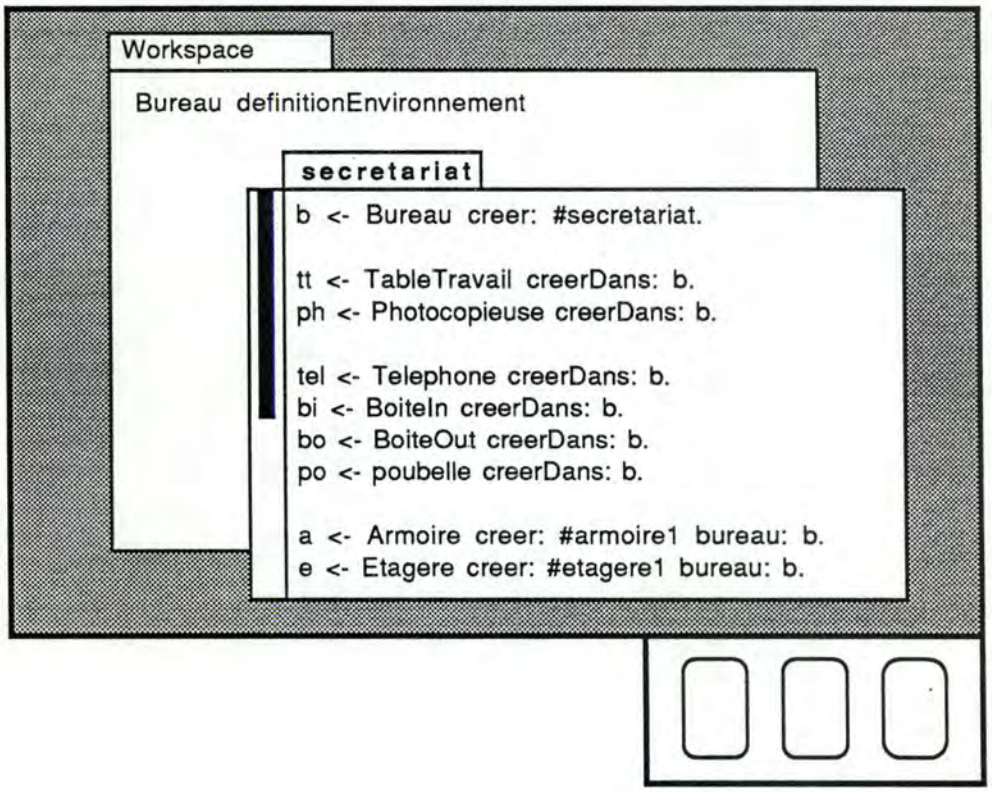


figure 3.3

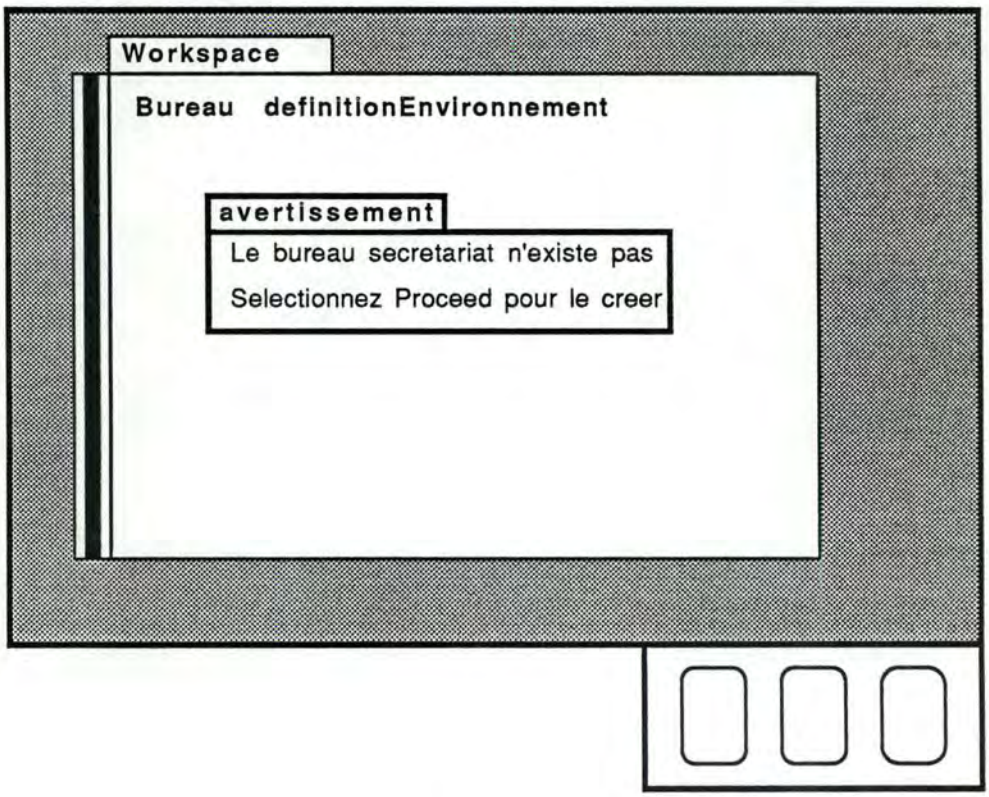


figure 3.4

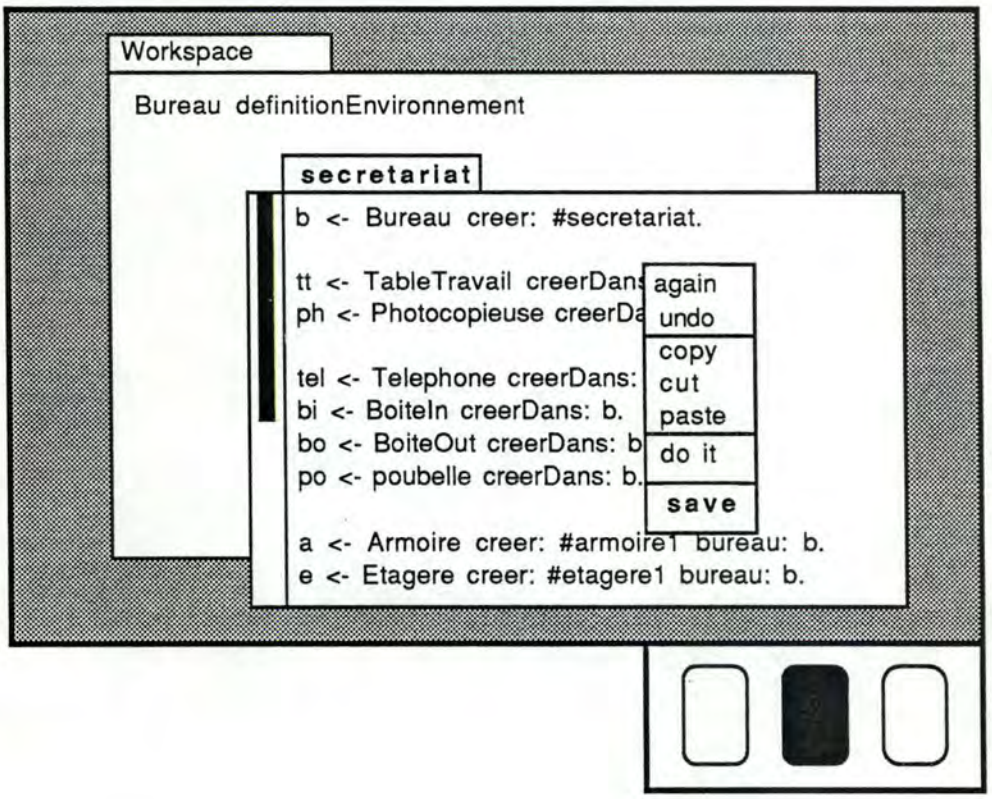


figure 3.5

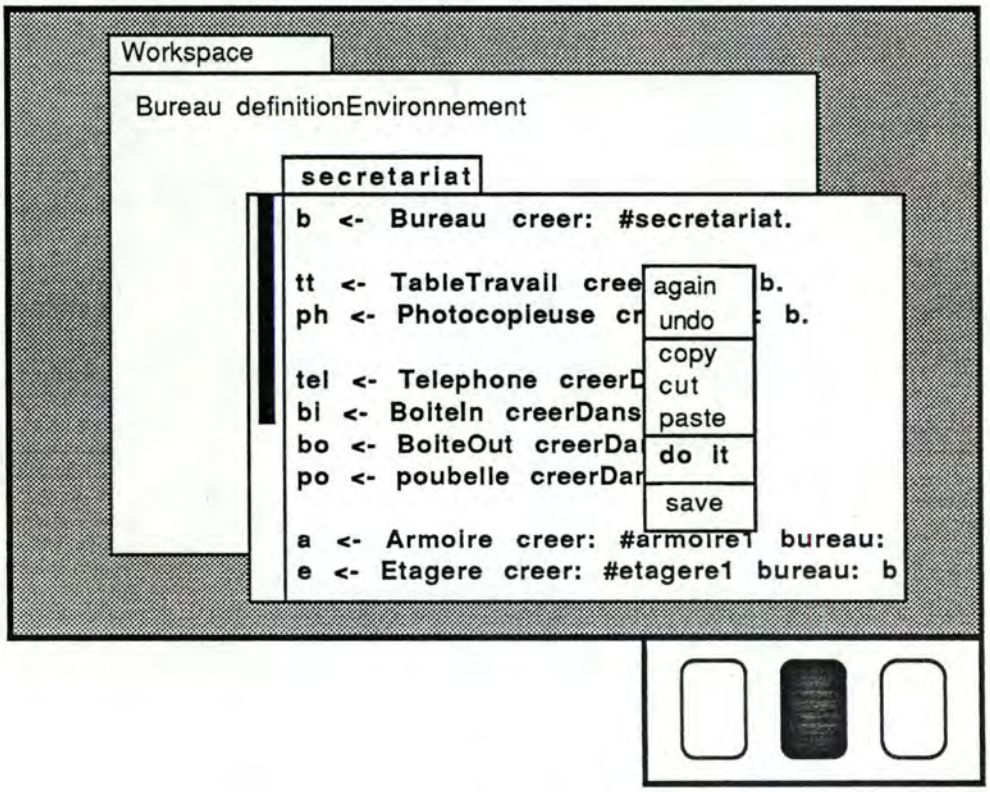


figure 3.6

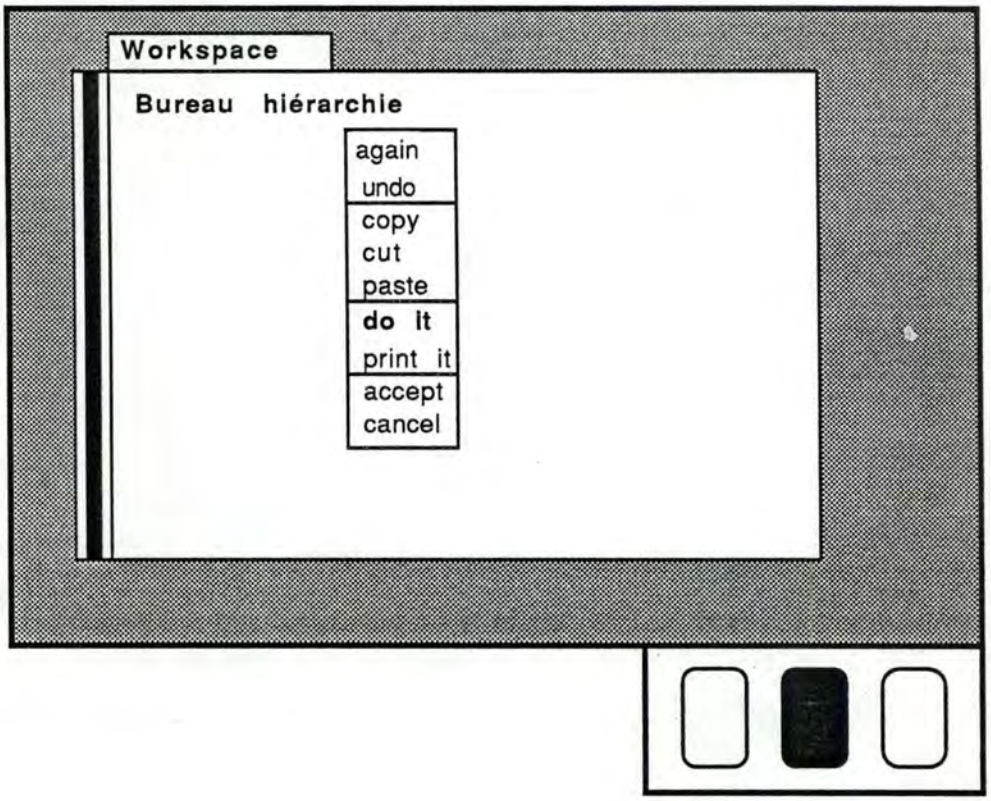


figure 3.19

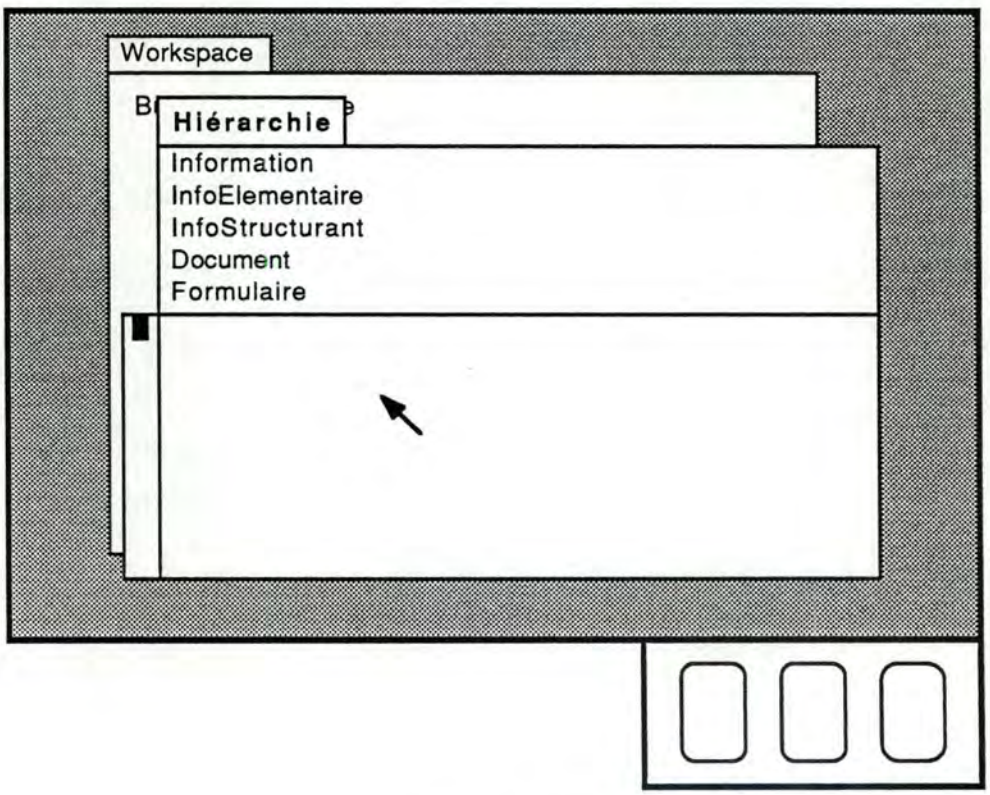


figure 3.20

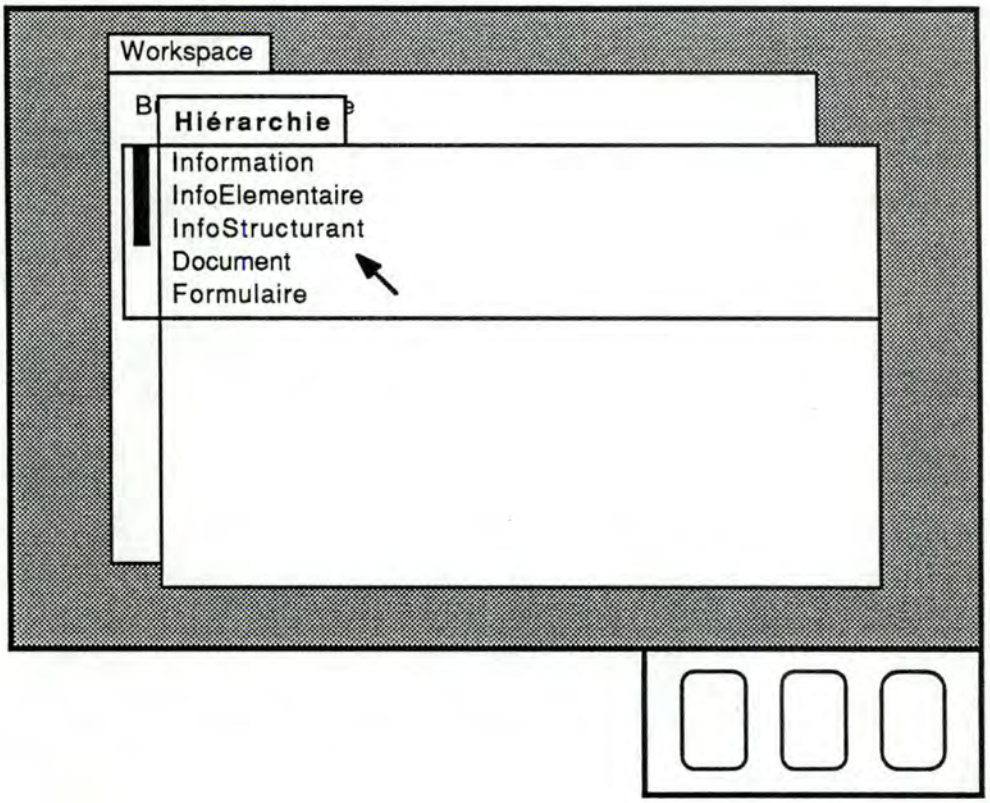


figure 3.21

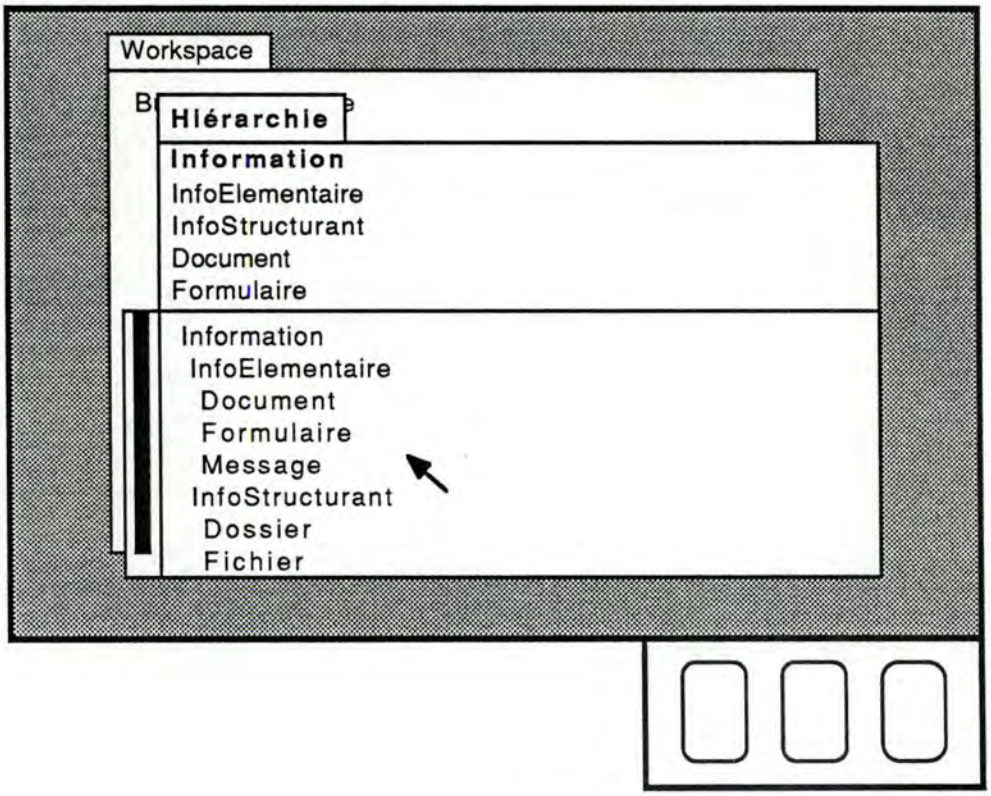


figure 3.22

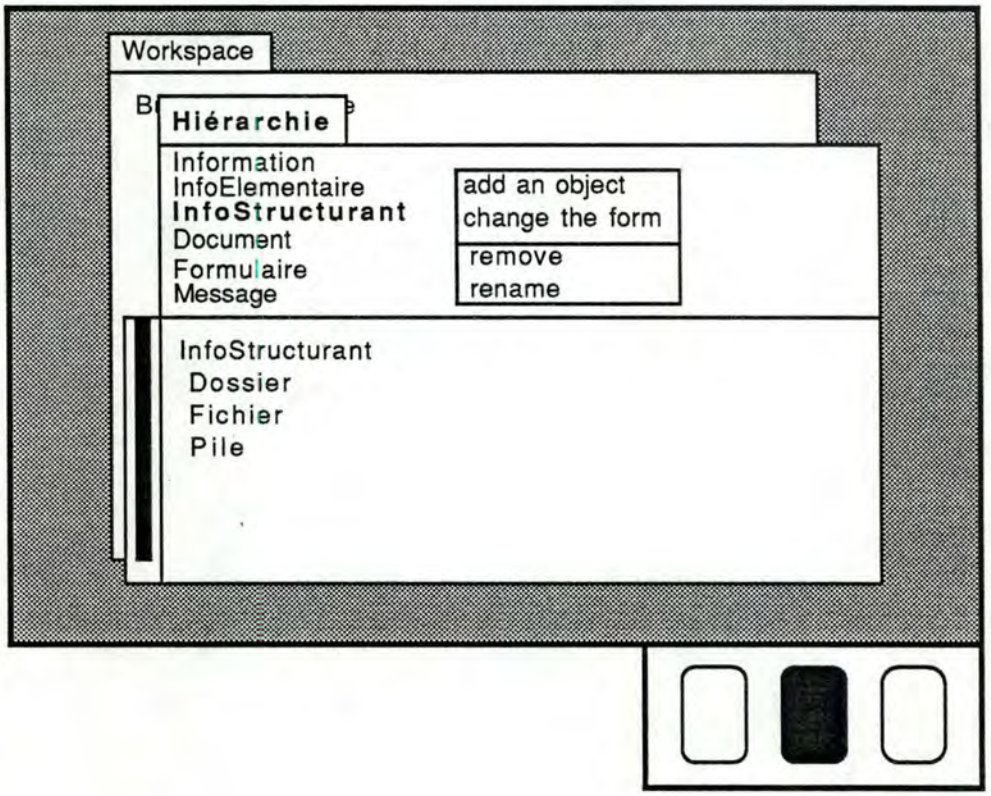


figure 3.23

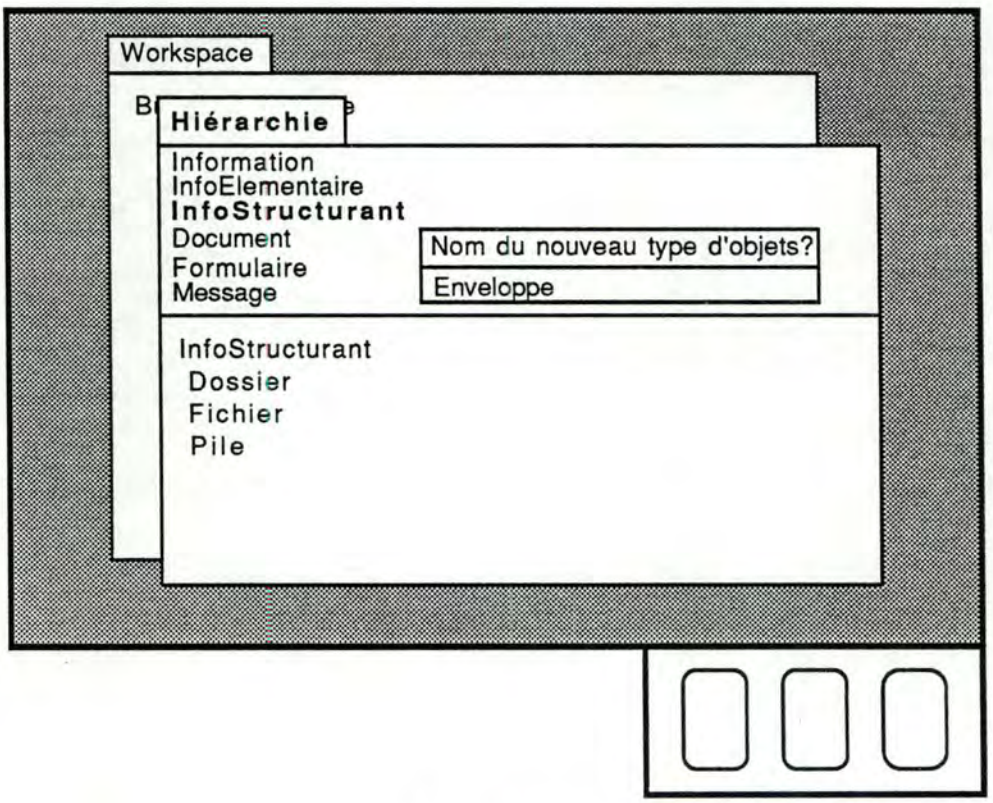


figure 3.24

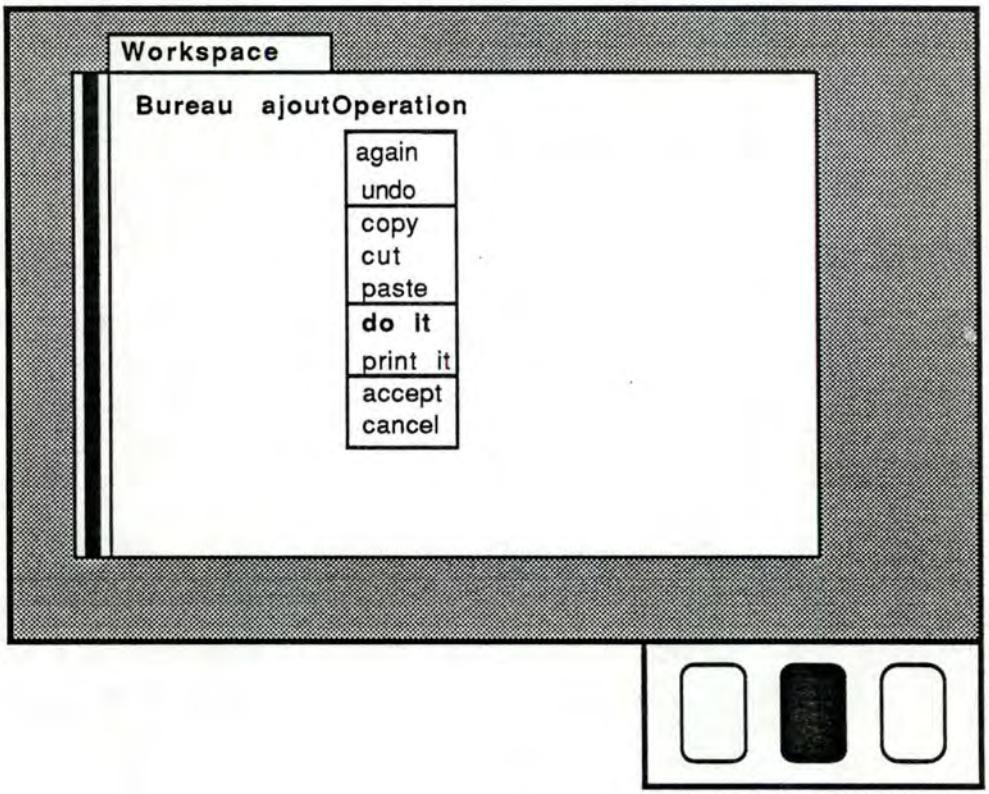


figure 3.25

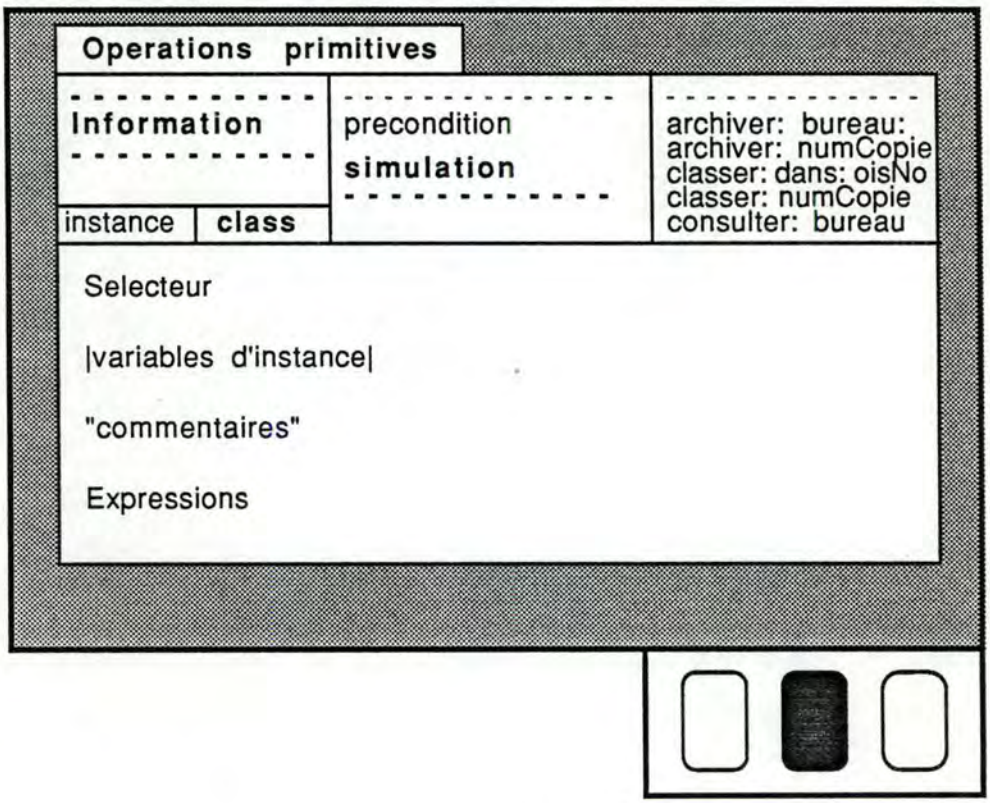


figure3.26